ESCUELA SUPERIOR DE INGENIERÍA

SEGUNDO CICLO EN INGENIERÍA INFORMÁTICA

ANALIZADOR DE SERVICIOS WEB
BASADOS EN WSDL 1.1
PARA PRUEBAS PARAMÉTRICAS

Cristina Jiménez Gavilán

Cádiz, Mayo 2011

ESCUELA SUPERIOR DE INGENIERÍA

SEGUNDO CICLO EN INGENIERÍA INFORMÁTICA

ANALIZADOR DE SERVICIOS WEB BASADOS EN WSDL 1.1 PARA PRUEBAS PARAMÉTRICAS

DEPARTAMENTO: LENGUAJES Y SISTEMAS INFORMÁTICOS

DIRECTORES DEL PROYECTO: JUAN JOSÉ DOMÍNGUEZ JIMÉNEZ Y ANTONIO GARCÍA DOMÍNGUEZ

AUTOR DEL PROYECTO: CRISTINA JIMÉNEZ GAVILÁN

Cádiz, Mayo 2011

Fdo.: Cristina Jiménez Gavilán

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1

Introducción

1.1. Antecedentes

El flujo de conocimiento que dirige una empresa u organización en cualquier sector de negocio cambia de forma constante como respuesta a influencias externas e internas. Cada área de negocio de la empresa origina unos requisitos de negocio que son implementados mediante procesos documentados, es lo que se conoce como *capa de negocio*. Por otro lado, existe un conjunto de aplicaciones que automatizan dichos procesos de negocio basándose en distintas soluciones tecnológicas y que conforman la *capa de aplicación*. Ésta última se caracteriza por utilizar tecnologías heterogéneas, por tener un origen diverso (la propia organización u otros proveedores) y por estar sujeta a restricciones (técnicas, de seguridad, etc.).

En este mundo empresarial tan cambiante lo ideal sería minimizar las dependencias entre la capa de negocio y la capa de aplicación para desacoplar el negocio de la tecnología y, de este modo, minimizar el impacto del cambio en alguna de ellas.

En la actualidad, los Servicios Web (*Web Service* o WS en inglés) y las arquitecturas orientadas a servicios parecen ser una de las claves para lograr este objetivo. De ahí que en los últimos años se haya detectado un importante y sostenido aumento en la inversión que las empresas dedican al desarrollo de software basado en servicios y en concreto, al desarrollo de composiciones de servicios.

1.1.1. Servicios Web

Existen múltiples definiciones sobre lo que son los Servicios Web, lo que muestra su complejidad a la hora de dar una adecuada que englobe todo lo que son e implican.

Una forma sencilla de definirlos sería hablar de ellos como un conjunto de aplicaciones o de tecnologías con capacidad para interoperar en la Web, intercambiando datos entre sí, con el objetivo de ofrecer unos servicios.

El W3C (World Wide Web Consortium) define un Servicio Web como una aplicación software identificada por un URI cuyas interfaces se pueden definir, describir y descubrir mediante documentos XML [16].

El auge de los WS se debe a que permiten la interoperación de sistemas distribuidos heterogéneos con independencia de las plataformas hardware y software empleadas.

Los WS son en definitiva una implementación del paradigma de Arquitectura Orientada a Servicios, SOA de aquí en adelante. Este paradigma define un nuevo estilo de arquitectura abstracta que no está ligado a ninguna tecnología en particular y en la que el *servicio* es el elemento atómico. Se basa en la creación de un conjunto de servicios de diferente granularidad entre los procesos de negocio y las aplicaciones, cumpliendo los siguientes principios:

1. Toda función, sea simple o compuesta, es descrita como un servicio.

2. Todo servicio es independiente de los demás servicios y sólo se comunica a través de su interfaz.

3. Toda interfaz de servicio puede ser invocada con un identificador único, independientemente de que sea local o remoto.

De estos principios se derivan dos conceptos vitales en el ámbito de los Servicios Web:

La composición de servicios se refiere a la reutilización y combinación de servicios de forma automatizable. Permite, desde la invocación de servicios condicionada al contenido de la petición, a la transformación del mensaje para facilitar la convivencia de versiones o la adaptación de formatos.

Es también conocida como orquestación o coreografía. Para distinguir entre estos dos términos primero se debe analizar las características de cada uno de ellos.

En la **orquestación**, existe un control centralizado que dirige las actividades de un conjunto de partes, en este caso, Servicios Web. Cada uno de estos WS recibe el pedido de ejecución de un *orquestador* y probablemente devolverá una respuesta del trabajo realizado. Un *orquestador* es un tipo de servicio particular que es capaz de coordinar los servicios realizando distintas invocaciones a los mismos.

Una **coreografía**, en cambio, no posee un coordinador central, sino que sus partes se organizan para llevar a cabo la tarea, de acuerdo a una colaboración de pares (*peer-to-peer*). La perspectiva es neutral para cualquiera de los WS participantes en cualquier interacción, y la ejecución y el control central son responsabilidad de los participantes en sí mis-

mos. En términos generales, la coreografía es menos restrictiva que la orquestación.

Una analogía que podría considerarse es respecto a un semáforo y una rotonda. El semáforo representa la orquestación, debido a que el control de los eventos que se suceden se lleva a cabo de manera centralizada. La rotonda representa la coreografía, ya que cada participante sigue un conjunto de reglas preestablecidas.

WS-BPEL (Web Services Business Process Execution Language) 2.0 es un ejemplo de orquestación, en la que el "director de orquesta" es el motor BPEL utilizado. Como ejemplo de coreografía podemos citar WS-CDL (Web Services Choreography Description Language).

El descubrimiento de un WS es un proceso que consiste en buscar y consultar descripciones de Servicio Web como paso previo a tener acceso al mismo. De este modo, los clientes de Servicios Web pueden tener conocimiento de la existencia de un WS, sus capacidades y la forma correcta de interactuar con él.

En el momento de llevar esta teoría a la práctica nos encontramos con una infraestructura en la que participan tres actores principales: un *proveedor de servicios*, un *agente de servicios* y un *consumidor de servicios*.

- *Proveedor de servicios*: crea un Servicio Web y publica su interfaz y la información de acceso al registro *agente del servicio*. Decide qué servicios exponer, qué política de seguridad establecer, cómo fijar el precio del servicio, etc.

- *Agente de servicios*: es el responsable de construir la interfaz de Servicio Web y la implementación accesible a cualquier solicitante potencial

de servicio. Concreta detalles tales como qué público podrá acceder al servicio y la cantidad de información que se le ofrecerá.

- *Consumidor de servicios*: solicita o es cliente de un servicio. Localiza entradas en el registro agente realizando diferentes búsquedas y se une al proveedor de servicios con el fin de invocar uno de sus WS.

Son numerosas las ventajas que ofrecen este tipo de arquitectura. Caben destacar las siguientes:

1. Diversifican las oportunidades de negocio al facilitar que aparezcan escenarios de libre intercambio de servicios normalizados.

2. Son arquitecturas acopladas de manera muy débil, lo que facilita la operación entre plataformas con distinto hardware, sistema operativo o que empleen distintos lenguajes de programación en sus aplicaciones.

3. Los servicios son fáciles de reutilizar y reemplazar, lo que produce una reducción de costes, especialmente del de mantenimiento.

4. Es posible lograr un mayor control sobre el funcionamiento de los procesos remotos cuando se emplean las tecnologías adecuadas.

5. Pueden funcionar síncrona o asíncronamente, según las necesidades.

Pero realmente SOA y los servicios son conceptos que existen ya desde hace tiempo, de hecho SOA no deja de ser una arquitectura distribuida. Lo que sí es nuevo es el enfoque de negocio que incorpora, así como la existencia de estándares y de tecnologías capaces de implementar estos conceptos y que hacen posible esta circulación de información. Los WS se construyen a partir de varios estándares abiertos basados en XML, proporcionando así una integración perfecta y la interoperabilidad entre aplicaciones software a través la red

(véase [50]). Algunos de estos estándares ya existían, pero se utilizaban en otros ámbitos. Entre ellos, podemos destacar los siguientes (ver figura 1.1):

- XML (eXtensible Markup Language) [60]: sirve para estructurar, almacenar e intercambiar información. Deja a los diseñadores crear sus propias "etiquetas", habilitando la definición, transmisión, validación e interpretación de datos entre organizaciones y/o aplicaciones diferentes.
- SOAP (Simple Object Access Protocol) [29]: se trata de un protocolo basado en XML, que especifica el formato de los mensajes, permitiendo la interacción entre varios dispositivos. Tiene la capacidad de transmitir información compleja.
- HTTP (HyperText Transfer Protocol) [61]: es uno de los protocolos a través del cual pueden ser transmitidos los mensajes SOAP que intervienen en la comunicación del WS.
- WSDL (Web Services Description Language) [58]: permite que un servicio y un cliente establezcan un acuerdo en lo que se refiere a los detalles de la sintaxis y los mecanismos de intercambio de mensajes, a través de un documento procesable por dispositivos. Puede verse como la "interfaz" o contrato que proporciona el *proveedor del servicio* al *consumidor del servicio* con la estructura de los mensajes y la localización física del servicio. WSDL es muy abierto, por lo que han surgido especificaciones como el WS-I 1.1 Basic Profile que lo acotan para mejorar la interoperabilidad. A su vez WSDL emplea otro estándar, XML Schema, que se utiliza para describir el formato esperado en los mensajes a intercambiar.
- UDDI (Universal Description Discovery and Integration) [45]: es la especificación de OASIS (Organization for the Advancement of Structured

Information Standards) que define una forma de publicar y descubrir información sobre los WS. Conforma una especie de directorio en el cual podemos encontrar los WS publicados y publicar los que desarrollemos, como unas “páginas amarillas”.

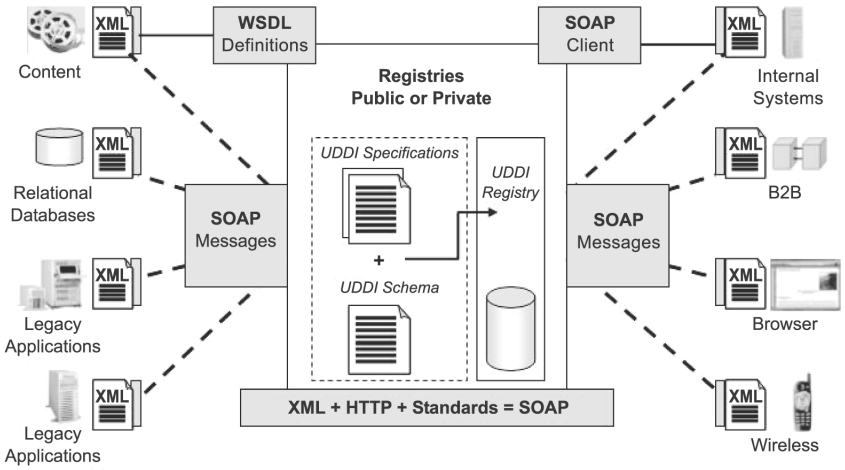

Figura 1.1: Uso de XML, SOAP, WSDL y UDDI en los Servicios Web

1.1.2. Pruebas

El gran auge que están alcanzando en los últimos años los Servicios Web y las composiciones de éstos hace necesario prestar especial atención a la prueba de este tipo de software. La fase de pruebas en el desarrollo de SOA debe abordarse de manera distinta a como se hace en el desarrollo de software tradicional, ya que estos sistemas poseen ciertas peculiaridades con respecto al mismo. De [6] hemos extraído aquellas que dificultan el proceso de pruebas:

1. Escaso conocimiento del código de los servicios y su estructura: esto se debe a que tanto los usuarios, como los otros servicios, utilizan el servicio en cuestión a través de su interfaz.

2. Dinamismo y adaptabilidad: son un arma de doble filo cuando de pruebas se trata ya que es casi imposible poder probar todos los posibles casos que se pueden dar.

3. Falta de control: resulta complicado controlar un servicio debido a que como tal, no se integra en el sistema, sino que se ejecuta en una infraestructura independiente que controla el proveedor del servicio.

4. El coste de la prueba: se encarece al tener que ejecutar el servicio remotamente, en la mayoría de los casos.

Para abordar estos retos deben realizarse investigaciones conducentes a determinar en qué medida los procesos de prueba válidos para otro tipo de software pueden ser aplicados a software basado en servicios y qué nuevas técnicas específicas serían adecuadas para este tipo de sistemas.

En concreto, las técnicas de caja blanca y, más precisamente, las técnicas de *prueba de mutaciones* pueden ser elementos clave en el desarrollo de nuevas estrategias enfocadas a la prueba de composiciones de servicios web.

Por otro lado, hemos dicho que el coste de las pruebas de servicios web puede ser alto. Una manera eficaz de reducirlo es mediante el uso de técnicas que permitan su *automatización*. Rice en [49] enumera y explica los diez retos más importantes en la automatización del proceso de pruebas. Podría resumirse en la falta de herramientas y de compatibilidad entre las existentes (bien por su elevado precio o bien porque las existentes no se ajusten al propósito o entorno para el que se necesitan) y la falta de un proceso de gestión de la configuración de calidad. Todo ello se debe en gran parte al rechazo, tanto del cliente como de la organización, a invertir en pruebas.

El grupo de investigación UCASE [1] de Ingeniería del Software de la Universidad de Cádiz incluye entre sus líneas de investigación la mejora automatizada de las pruebas de software. Dentro del ámbito de los Servicios Web, el grupo ha desarrollado `GAmera`, una herramienta que localiza posibles fallos que las pruebas iniciales no podrían detectar. El siguiente paso en esta línea es la generación de pruebas que puedan detectar estos fallos, extendiendo el conjunto inicial. Sin embargo, en el ámbito de los Servicios Web, como hemos visto en el apartado anterior, suelen participar tecnologías y estándares muy diversos, por lo que hay que considerar una amplia variedad de restricciones técnicas para generar casos de prueba válidos.

La alumna del presente Proyecto Fin de Carrera (PFC, en adelante) colabora con el grupo de investigación trabajando en esta línea. Fruto de esta colaboración, surge la idea de crear el *ServiceAnalyzer*, una aplicación que se ocupará de abstraer al futuro generador de casos de prueba de `GAmera` de la mayoría de estos detalles técnicos. Para ello, se implementará un analizador de las declaraciones de las interfaces de los Servicios Web en el lenguaje WSDL 1.1. El analizador producirá un catálogo de plantillas en el lenguaje Apache Velocity que serán capaces de generar todas sus posibles entradas y salidas válidas. Las plantillas se parametrizarán en base a una serie de variables, usando un sistema de tipos simplificado cuyos valores sean más fáciles de generar que los del complejo sistema de tipos de WSDL y XML Schema.

Los mensajes generados respetarán las restricciones impuestas por las declaraciones de las operaciones en WSDL, las declaraciones de tipos XML Sche-

[1] El grupo de investigación UCASE de Ingeniería del Software se creó en el año 2011, dentro del Plan Andaluz de Investigación (PAI), que le asigna la referencia TIC-025. Está integrado por un grupo de profesores del Departamento de Lenguajes y Sistemas Informáticos y tiene su sede en la Universidad de Cádiz. Surge a partir de separación de las líneas de investigación de otro grupo, SPIFM (Software Process Improvement and Formal Methods) que, como su nombre indica, se centra en los campos de la mejora del proceso software y en los métodos formales, teniendo como objetivo final contribuir a mejorar la calidad del software.

ma, y las restricciones para interoperabilidad del WS-I Basic Profile 1.1. De este modo, se separa la generación de casos de prueba de los detalles de WSDL y SOAP, liberando el proceso de generación de casos de prueba de las restricciones de las diferentes tecnologías que hacen que el proceso sea incómodo y propenso a errores difíciles de depurar.

El grupo de investigación utilizará las plantillas generadas como parte de la entrada de `BPELUnit`, el marco de pruebas unitarias de composiciones de Servicios Web basadas en WS-BPEL 2.0 que está integrado en `GAmera`.

No obstante, el proyecto podrá ser usado en cualquier contexto de pruebas de composiciones de servicios web cuyas interfaces estén descritas mediante WSDL 1.1.

Se ha utilizado la versión 1.1 de WSDL en lugar de la 2.0 porque es la que utilizan las plataformas con las que el programa deberá interactuar (en concreto, `ActiveBPEL`, `BPELUnit` y `MuBPEL`) como parte de un proyecto mayor del grupo. Además, a pesar de que WSDL 2.0 surgió hace varios años, es una versión que aún no es soportada por muchas tecnologías, debido a que presenta bastantes cambios frente a la versión anterior.

1.2. Objetivos

- Implementar un analizador de Servicios Web basados en WSDL 1.1 para pruebas paramétricas. Se necesitará:
  - Definir un sistema de tipos simplificado que facilite la generación de valores con respecto al complejo sistema de tipos que presentan WSDL y XML Schema.
  - Generar plantillas para todas las posibles entradas, salidas y fallos, usando un lenguaje de plantillas. El hecho de utilizar un lenguaje

de plantillas facilita el uso de variables en las mismas, de modo que, pueden ser empleadas para más de un caso de prueba, en función del valor que se le asigne a las variables.

- Suministrar las declaraciones de tipo de las variables de las plantillas en el sistema de tipos simplificado creado.

- Generar un conjunto de casos de prueba consistente que, además de garantizar en cierto grado la corrección del programa, facilite al grupo de investigación la implementación de futuras modificaciones o extensiones. El trabajo tendrá continuidad en proyectos de investigación en desarrollo dentro del departamento y en coordinación con otras universidades.

- Crear los catálogos de plantillas de todas las composiciones de servicios con las que trabaja el grupo UCASE, obteniendo un resultado satisfactorio en todas aquellas que respeten las restricciones del WS-I Basic Profile 1.1.

- Utilizar una herramienta que proporcione informes de métricas de calidad del código y, a partir de ellos, refinar el mismo de acuerdo a las prácticas recomendadas que comprueban esas herramientas.

- Publicar el código fuente y la documentación asociada a través de un repositorio público con licencia libre.

1.3. Alcance

El proyecto se compone de dos productos:

- El producto principal es *ServiceAnalyzer*, un analizador de las declaraciones de las interfaces de Servicios Web en el lenguaje WSDL 1.1 que crea

un catálogo de plantillas en el lenguaje Apache Velocity, parametrizadas en base a una serie de variables, que pretende facilitar la generación de casos de prueba para las composiciones que empleen dichas interfaces.

- *WSDL2XSDTree*, una aplicación que construye un árbol de descripciones XML Schema a partir de un fichero WSDL suministrado por el usuario. *ServiceAnalyzer* utiliza internamente este producto para organizar y aislar las descripciones de tipos que necesita consultar para construir las declaraciones de tipos de las variables del catálogo.

Ambos productos dispondrán de su propia documentación, tanto manuales de instalación y uso para usuarios como, documentación del código para desarrolladores. Asimismo, se suministrará el conjunto de pruebas unitarias que se empleen para la fase de pruebas de los dos productos.

Aunque son dos los productos principales del PFC, cabe destacar que se ha llevado a cabo asimismo la reingeniería de diversas composiciones WS-BPEL para el desarrollo de las pruebas, así como la construcción de ficheros BPTS (BPELUnit Test Suite) que amplían el conjunto de casos de pruebas. A continuación, se resumen estas tareas:

- Nuevos casos de pruebas para el ejemplo del LoanApproval.
- Transformación del ejemplo del ShippingService en una composición ejecutable asíncrona.
- Correcciones en el ejemplo del LoanApprovalRPC para que cumpla con las restricciones del WS-I Basic Profile 1.1.

1.3.1. Limitaciones del proyecto

Por limitaciones de tiempo, en las traducciones de los tipos utilizados en los mensajes a intercambiar, se han descartado aquellos cuyas descripciones

contengan las siguientes restricciones XML Schema:

- Union
- Choice
- All
- Mixed
- Wildcard
- Whitespace

El propósito del estándar XML Schema [59] es definir la estructura de los documentos XML que estén asignados a tal esquema y los tipos de datos válidos para cada componente(elemento, atributo). En este sentido las posibilidades de control sobre la estructura y los tipos de datos son muy amplias, por lo que resulta complicado abarcarlas todas.

En la práctica, por razones de seguridad, rara vez se usan los indicadores XML Schema anteriormente citados para definir mensajes a intercambiar entre Servicios Web, ya que no suele desearse esa "flexibilidad" en el intercambio de información. Es por esto por lo que se ha pospuesto su implementación a futuras versiones de la aplicación.

1.3.2. Licencia

En cuanto a la licencia del proyecto, se desea que la aplicación `Service-Analyzer` tenga el mejor uso público posible y la mejor manera para alcanzar esto es publicar el programa bajo una licencia de software libre. La presente documentación también queda bajo licencia libre.

Entre las numerosas ventajas del Software Libre que nos han llevado a tomar esta decisión podemos citar:

- Escrutinio público: Puesto que muchas personas van a tener acceso al código fuente, el software libre estará expuesto a un proceso continuo y muy dinámico de corrección de errores, sin tener que esperar a que el proveedor del software saque una nueva versión.

- Independencia del proveedor: Al disponer del código fuente, cualquier persona puede continuar ofreciendo soporte, desarrollo u otro tipo de servicios para el software, es decir, no estamos supeditados a las condiciones del mercado de nuestro proveedor.

- Garantía de continuidad: dado que cualquier otra persona puede continuar desarrollándolo, mejorándolo o adaptándolo, el software libre puede seguir siendo usado incluso después de que haya desaparecido la persona que lo creó originalmente.

- Manejo de la lengua: cualquier persona cualificada puede traducir y adaptar un software libre a cualquier lengua. Una vez traducido el software libre puede presentar errores de tipo ortográfico o gramatical, los cuales pueden ser subsanados con mayor rapidez por una persona capacitada.

- Mayor seguridad y privacidad: Los sistemas de almacenamiento y recuperación de la información son públicos por lo que cualquier persona puede ver y entender cómo se almacenan los datos en un determinado formato o sistema. De este modo, existe una mayor dificultad para introducir código malicioso.

- Ahorro en costes: sobre todo disminuye el coste de adquisición ya que, al otorgar la libertad de distribuir copias, se puede ejercer con la compra de una única licencia y no con tantas como equipos posea (caso de la

mayoría de los programas propietario). También suelen disminuir los costes de soporte.

En concreto, se ha decidido publicar *ServiceAnalyzer* y *WSDL2XSDTree* bajo la Apache Software License (ASL) 2.0 [19]. La toma de la decisión se ha basado en que esta licencia se ajusta a las requerimientos del Proyecto y además, es usada por varias de las herramientas empleadas y compatible con la licencia del resto.

Las licencias de las principales dependencias utilizadas en los productos de este PFC pueden verse en las tablas 1.1 [2] y 1.2. En cada una de ellas se detalla el árbol de dependencias del producto, con el ámbito de cada dependencia y la licencia a la que está sujeta. El ámbito o alcance de una dependencia se refiere a la fase en la que se utilizan. Se pueden considerar tres tipos de alcance:

- Compilación: son necesarias para compilar y ejecutar la aplicación.
- Ejecución: son imprescindibles a la hora de ejecutar la aplicación.
- Prueba: únicamente se requieren si se va proceder a la compilación y ejecución de los casos de prueba.

Todas las dependencias de compilación y ejecución de *WSDL2XSDTree* son a su vez dependencias transitivas de *ServiceAnalyzer*, por lo que se ha evitado repetirlas en la tabla 1.2.

[2] La licencia CPL (Common Public License) ha sido sustituida por la licencia EPL (Eclipse Public License) mediante un acuerdo entre IBM y la Fundación Eclipse, quedando ésta última como el Administrador del Acuerdo en la EPL y eliminado la sentencia "If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed." de la cláusula de patentes (ver [31], sección 7).

Dependencia	Ámbito	Licencia
JUnit 4.8.1	Prueba	Common Public License V 1.0
WSDL4J 1.6.2	Compilación	Common Public License
XML Utils	Compilación	Apache Software License V 2.0
. . . Saxon 9.1.0.1	Compilación	Mozilla Public License V 1.0
. . . XMLBeans 2.3.0	Compilación	Apache Software License V 2.0
. . . . . StAX API 1.0.1	Compilación	Apache Software License V 2.0

Tabla 1.1: Licencias de las dependencias de WSDL2XSDTree

Dependencia	Ámbito	Licencia
Apache Velocity 1.6.4	Prueba	Apache Software License V 2.0
JOpt Simple 3.2	Compilación	MIT License
JUnit 4.8.1	Prueba	Common Public License V 1.0
SLF4J API 1.6.1	Compilación	MIT License
SLF4J log4j-12 binding 1.6.1	Ejecución	MIT License
. . . Apache Log4j 1.2	Ejecución	Apache Software License V 2.0
. . . SLF4J API 1.6.1	Ejecución	MIT License
WSDL2XSDTree	Compilación	Apache Software License V 2.0

Tabla 1.2: Licencias de las dependencias de ServiceAnalyzer

1.4. Visión general

El presente documento se encuentra organizado en ocho capítulos. En el capítulo actual, “Introducción”, se ha presentado brevemente el origen y la funcionalidad de la aplicación *ServiceAnalyzer*.

En el capítulo 2, “Desarrollo del calendario”, vamos a tratar la planificación temporal del desarrollo del proyecto, su seguimiento y el porcentaje de esfuerzo dedicado a las diferentes fases del proceso de desarrollo.

El capítulo número 3 se denomina “Descripción general del proyecto”. En esta sección se dará una visión general del proyecto, ampliando el contenido del capítulo de introducción.

En el capítulo 4, “Desarrollo del proyecto”, se describen los aspectos relacionados con las actividades que han sido necesarias para realizar el análisis y

especificación de los requisitos, el diseño, la codificación, la integración y las pruebas de la aplicación software creada.

El siguiente capítulo es el de “Resumen” que, como su propio nombre indica, sintetiza lo más destacable del trabajo realizado.

El capítulo 6 se llama “Conclusiones”. En este apartado se realiza una valoración global del trabajo abordado en el PFC.

El siguiente capítulo contienen un manual con las instrucciones de instalación y de uso y algunas pistas para un futuro desarrollador.

A continuación, se incluyen una serie de anexos que complementan la información recogida en los capítulos anteriormente descritos, como por ejemplo, la licencia de la presente documentación.

Finalmente, en “Referencias” se muestran los libros, los artículos y los principales enlaces electrónicos que se han utilizado durante la realización de este proyecto, tanto aquellos que se han empleado para el desarrollo de la aplicación, como aquellos que se han necesitado para la elaboración de la presente documentación.

1.5. Glosario

1.5.1. Acrónimos

API Application Programming Interface

ASL Apache Software License

AST Abstract Syntax Tree

BPEL Business Process Execution Language

BPTS BPELUnit Test Suite

CASE Computer Aided Software Engineering

CCPL Creative Commons Public License

CDDL Common Development and Distribution License

CI Continuous Integration

CPL Common Public License

CSV Comma-Separated Values

CVS Concurrent Versions System

DOM Document Object Model

DTD Document Type Definition

EPL Eclipse Public License

GMT Greenwich Mean Time

GNU GNU is Not Unix

GPL GNU General Public License

HTML Hyper Text Markup Language

HTTP HyperText Transfer Protocol

IBM International Business Machines

IDE Integrated Development Environment

IDL Interface Description Language

IM Instant Messaging

IEEE Institute of Electrical and Electronics Engineers

J2SE Java 2 Standard Edition

JAXB Java Architecture for XML Binding

MEC Ministerio de Educación y Ciencia

MEP Message Exchange Pattern

MIME Multipurpose Internet Mail Extensions

MOJO Maven Old Java Object

MTOM Message Transmission Optimization Mechanism

NASSL Network Accessible Service Specification Language

OASIS Organization for the Advancement of Structured Information Standards

OWL-S Ontology Web Language for Services

RPC Remote Procedure Call

PAI Plan Andaluz de Investigación

POJO Plain Old Java Object

POM Project Object Model

PRIS Pruebas en Ingenieria del Software

RSS Really Simple Syndication

SAX Simple API for XML

SCM Software Configuration Management

SCV Sistema de Control de Versiones

SDL Service Description Language

SGML Standard Generalized Markup Language

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

SLF4J Simple Logging Facade for Java

SMTP Simple Mail Transfer Protocol

SPIFM Software Process Improvement and Formal Methods

StAX Streaming API for XML

TDD Test-Driven Development

UDDI Universal Description Discovery and Integration

UML Unified Modeling Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

UTC Coordinate Universal Time

VTL Velocity Template Language

W3C World Wide Web Consortium

WS Web Services

WS-BPEL Web Services Business Process Execution Language

WS-CDL Web Services Choreography Description Language

WSDL Web Services Description Language

WSDL4J Web Services Description Language for Java

WS-I Web Services Interoperability Organization

WWW World Wide Web

XSD XML Schema Definition

XML eXtensible Markup Language

XPath XML Path Language

XQuery XML Query

XDM XML Data Model

XHTML eXtensible Hyper Text Markup Language

XP eXtreme Programming

XSLT eXtensible Stylesheet Language Transformations

XSL eXtensible Stylesheet Language

XSOM XML Schema Object Model

1.5.2. Definiciones

Pruebas de caja blanca Técnica de prueba que consiste en centrarse en la estructura interna (implementación) del programa para elegir los casos de prueba. Constituye el llamado enfoque estructural, frente al denominado enfoque funcional o de caja negra.

Pruebas de caja negra Técnica de prueba que consiste en centrarse en lo que se espera de un módulo, es decir, intenta encontrar casos en que el módulo no se atiene a su especificación. El probador se limita a suministrarle datos como entrada y estudiar la salida, sin preocuparse de lo que pueda estar haciendo el módulo por dentro. Las pruebas de caja negra se apoyan, por tanto, en la especificación de requisitos del módulo.

Pruebas de mutaciones Técnica de prueba de caja blanca del software basada en errores. Consisten en generar un gran número de programas, denominados *mutantes*, a partir de un programa original, cada uno de ellos con una única diferencia con respecto al mismo. Los mutantes se generan aplicando al código fuente los llamados *operadores de mutación*. Podemos distinguir dos tipos de mutantes en función del número de cambios introducidos:

Mutantes de primer orden Mutantes que contienen una única diferencia con respecto al programa original.

Mutantes de orden superior Mutantes que contienen más de una diferencia con respecto al programa original.

Operadores de mutación Conjunto de reglas predefinidas que introducen pequeños cambios sintácticos en un programa, sin que éste pierda su validez sintáctica, basados en los errores que suelen cometer habitualmente los programadores, o bien con la pretensión de forzar ciertos criterios de cobertura del código.

Mockup Servicio sustituto de un servicio real en una simulación, que implementa un comportamiento predefinido, y que suele emplearse en pruebas para reemplazar a un servicio costoso, al que no se puede acceder, o también para comprobar si la salida de un programa para un caso de

prueba es la esperada. No tienen lógica interna, limitándose a responder con mensajes predefinidos, o "fallando" si así se especifica.

Pruebas de regresión Es un tipo de prueba que comprueba que el software sigue funcionando de la forma esperada entre versión y versión.

Gestión de la Configuración del Software (SCM) Actividad de protección que se ocupa de gestionar la evolución y los cambios que se producen a lo largo de todo el ciclo de vida del software, así como de gestionar los costes y el esfuerzo necesarios para llevarla a cabo.

Paquete Debian Fichero comprimido que contiene los ficheros de la aplicación, cierta información adicional y un par de guiones específicos de apoyo. Su uso se limita a distribuciones GNU/Linux compatibles. A través de un paquete Debian, podemos instalar de forma muy sencilla cualquier software que necesitemos, sin tener que entrar en detalles de cómo está hecha o cómo se configura inicialmente la aplicación. Además, la información adicional contenida en dicho paquete permite al programa que se ocupa de gestionarlos (`dpkg`) mantener un control de las dependencias. Así, si instalamos una determinada aplicación, todas las bibliotecas requeridas por ésta serán automáticamente instaladas.

Forja Sitio web dedicado a hospedar proyectos de desarrollo de software. Normalmente, las forjas son gratuitas para proyectos de software libre y/o código abierto, pero también existen forjas para entornos comerciales. Los servicios proporcionados varían mucho entre forja y forja, pero como mínimo suelen tener espacio en algún sistema de control de versiones, un sistema de control de incidencias y un área en la que alojar ficheros para su descarga por los usuarios.

1.5.3. Otras tecnologías

1.5.3.1. WS-BPEL

WS-BPEL (véase [44]) es un lenguaje basado en XML que permite especificar el comportamiento de un proceso de negocio basado en interacciones con WS.

La estructura de un proceso WS-BPEL se divide en cuatro secciones:

1. Definición de relaciones con los socios externos, es decir, con el cliente que utiliza el proceso de negocio y con los WS a los que llama el proceso
2. Definición de las variables que emplea el proceso.
3. Definición de los distintos tipos de manejadores que puede utilizar el proceso (de fallos y de eventos).
4. Descripción del comportamiento del proceso de negocio. Se hace a través de las *actividades* que proporciona el lenguaje.

Todos los elementos definidos anteriormente son globales si se declaran dentro del proceso. También existe la posibilidad de declararlos de forma local mediante el contenedor *scope*, que permite dividir el proceso de negocio en diferentes ámbitos.

Los principales elementos constructivos de un proceso WS-BPEL son las actividades, que pueden ser de dos tipos:

Básicas Realizan una determinada labor.

Estructuradas Pueden contener otras actividades y definen la lógica de negocio.

A las actividades pueden asociarse un conjunto de atributos y de contenedores. Estos últimos pueden incluir diferentes elementos, que a su vez pueden

tener atributos asociados. Además, WS-BPEL permite realizar acciones en paralelo y de forma sincronizada. Por ejemplo, la actividad *flow* permite ejecutar un conjunto de actividades concurrentemente especificando las condiciones de sincronización entre ellas.

Como ya se dijo en §1.1.1, WS-BPEL es un ejemplo de orquestación, en la que el motor BPEL utilizado es el responsable del control. El motor que respeta el estándar WS-BPEL 2.0 es `ActiveBPEL`. Comparado con otros motores es bastante ligero, lo que reduce el tiempo necesario para ejecutar casos de prueba. La empresa ActiveVOS [2] se encarga de su mantenimiento y ofrece servicios y productos empresariales basados en él.

1.5.3.2. BPELUnit

`BPELUnit` [41] es la biblioteca de prueba unitaria WS-BPEL [44], creada por Philip Mayer. Puede usar cualquier motor que implemente WS-BPEL 2.0. Entre sus principales características está el uso de un formato XML (*fichero BPTS*) para describir los casos de prueba a ejecutar y la posibilidad de sustituir servicios externos con otros servicios (*mockups*) que los simulen desarrollando el comportamiento indicado en la especificación proporcionada por el usuario. Además, `BPELUnit` ofrece posibilidades para añadir envíos síncronos y asíncronos.

BPTS es la extensión de los ficheros que se emplean para especificar juegos de casos de prueba para `BPELUnit` [43]. De forma general, podemos decir que un fichero BPTS define qué proceso se va a probar y cómo. El formato del fichero está basado en XML y como tal, puede validarse (con un XML Schema, por ejemplo) e incluye un elemento raíz donde se especifican los espacios de nombres. Como podemos ver en la figura 1.2, el documento está estructura en dos partes fundamentales:

- ***Sección de despliegue***: contiene la información para el despliegue y la especificación de todos los *partners* (servicios a los que se conecta). Contiene los siguientes elementos:

  name Un nombre identificativo para el proceso.

  baseURL La URL sobre la que se desplegará.

  deployment La información sobre el despliegue del proceso.

  put El proceso a desplegar. Se especifica su nombre, el motor que lo ejecutará, el fichero WSDL que describe el servicio y el fichero `BPR` que contiene la composición.

  partner Los servicios externos a los que se conectará, junto a su descripción WSDL. Estos servicios externos pueden sustituirse por *mockups* cuando se especifique el caso de prueba. El atributo `name` debe identificar unívocamente al «partner», ya que será el nombre que se usará después para sustituirlo por uno simulado.

- ***Sección de casos de prueba***: contiene una serie de casos de prueba. Se especifican como un elemento raíz «testCases» en el que se anidan elementos «testCase» para cada caso de prueba.

  testCase El elemento para definir un caso de prueba. El atributo `name` es un identificador único del caso de prueba. Un caso de prueba puede basarse en uno anterior, definido con el atributo `basedOn`. Si el atributo `abstract` tiene el valor “true” entonces el caso de prueba no puede ser ejecutado. Por último, `vary` especifica si se debe ejecutar más de una vez el mismo caso de prueba cambiando tiempos de respuesta.

Cada caso de prueba contiene un elemento «clientTrack» y cero o varios «partnerTrack»:

clientTrack Define la petición que se enviará desde el cliente simulado.

partnerTrack Se utiliza para definir un *mockup*. La estructura es prácticamente idéntica al «clientTrack». El atributo `name` define qué servicio externo está siendo reemplazado. Cada caso de prueba debe contener un único «partnerTrack» por cada «partner».

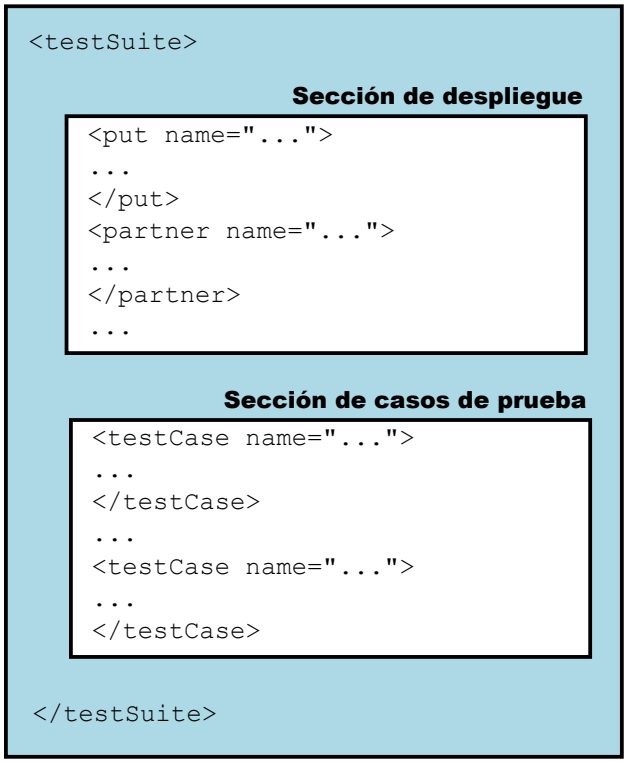

Figura 1.2: Estructura de un fichero BPTS

Tanto los «clientTrack» como los «partnerTrack» pueden contener una secuencia de actividades soportadas por `BPELUnit`:

sendOnly Esta actividad envía un mensaje simple. En el este bloque los datos incluidos dentro de «data» se enviarán al proceso. Estos datos seguirán la estructura definida por el propio proceso. El atributo `fault` a "false" indicará que no debe simularse un fallo, y a "true" que sí.

receiveOnly Esta actividad espera por un mensaje simple y lo verifica. Este bloque se evaluará cuando el proceso responda al cliente. Con «condition» pueden comprobarse si ciertas condiciones esperadas en el mensaje de respuesta se cumplen, si esta condición no se evalúa como verdadera para la actividad no pasará la prueba.

sendReceive En el caso de «clientTrack» el cliente del proceso mandará una petición síncrona. Los atributos especifican a qué servicio y puerto se enviará la petición, y qué operación se usará.

receiveSend Espera un mensaje simple, lo verifica, y envía una respuesta de vuelta sincronizada.

receiveSendAsynchronous Se reciben primero los datos, y se responde de forma asíncrona, por tanto, el proceso cliente no espera la respuesta.

sendReceiveAsynchronous Envía un mensaje simple, espera una respuesta asíncrona, y verifica la respuesta.

Listado 1.1: Ejemplo de un fichero *.bpts*

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <tes:testSuite
3     xmlns:esq="http://xml.netbeans.org/schema/Loans"
4     xmlns:ap="http://j2ee.netbeans.org/wsdl/ApprovalService"
5     xmlns:as="http://j2ee.netbeans.org/wsdl/AssessorService"
6     xmlns:sp="http://j2ee.netbeans.org/wsdl/LoanService"
```

```
7     xmlns:pr="http://enterprise.netbeans.org/bpel/N6_ServicioPrestamo/
          LoanApprovalProcess"
8     xmlns:tes="http://www.bpelunit.org/schema/testSuite">
9
10  <tes:name>LoanServiceTest</tes:name>
11  <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
12
13  <tes:deployment>
14    <tes:put name="LoanApprovalProcess" type="activebpel">
15      <tes:wsdl>LoanService.wsdl</tes:wsdl>
16      <tes:property name="BPRFile">LoanApprovalDoc.bpr</tes:property>
17    </tes:put>
18    <tes:partner name="assessor" wsdl="AssessorService.wsdl"/>
19    <tes:partner name="approver" wsdl="ApprovalService.wsdl"/>
20  </tes:deployment>
21
22  <tes:testCases>
23    <tes:testCase name="LargeAmount" basedOn="" abstract="false" vary="false">
24      <tes:clientTrack>
25        <tes:sendReceive
26            service="sp:LoanServiceService"
27            port="LoanServicePort"
28            operation="grantLoan">
29
30          <tes:send fault="false">
31            <tes:data>
32              <esq:ApprovalRequest>
33                <esq:amount>150000</esq:amount>
34              </esq:ApprovalRequest>
35            </tes:data>
36          </tes:send>
37
38          <tes:receive fault="false">
39            <tes:condition>
```

```
40                <tes:expression>esq:ApprovalResponse/esq:accept</tes:expression>
41                <tes:value>'true'</tes:value>
42              </tes:condition>
43            </tes:receive>
44          </tes:sendReceive>
45        </tes:clientTrack>
46
47        <tes:partnerTrack name="approver">
48          <tes:receiveSend
49              service="ap:ApprovalServiceService"
50              port="ApprovalServicePort"
51              operation="approveLoan">
52            <tes:send fault="false">
53              <tes:data>
54                <esq:ApprovalResponse>
55                  <esq:accept>true</esq:accept>
56                </esq:ApprovalResponse>
57              </tes:data>
58            </tes:send>
59            <tes:receive fault="false"/>
60          </tes:receiveSend>
61        </tes:partnerTrack>
62
63        <tes:partnerTrack name="assessor"/>
64
65      </tes:testCase>
66                  ...
67    </tes:testCases>
68
69  </tes:testSuite>
```

En el listado 1.1 podemos observar la estructura de un fichero BPTS mediante un fragmento de un fichero de pruebas de la típica composición de

WS-BPEL consistente en la aprobación de un préstamo (*Loan Approval*). Esta composición recibe un mensaje de un cliente que solicita una cierta cantidad de dinero. Dependiendo de la cantidad solicitada, el proceso WS-BPEL invoca al WS asesor (*asessor*) cuando la cantidad que se solicita es menor o igual a 10000, o al WS aprobador (*approver*), en otro caso. La salida del WS asesor se corresponde con el nivel de riesgo del cliente. Si el riesgo es bajo, se concede el préstamo, en caso contrario la petición se envía al WS aprobador, el cuál toma la decisión final de aceptación del préstamo.

Hasta la línea 8 tenemos el elemento raíz y la definición de los espacios de nombre. En este caso, el prefijo `tes` está asociado al espacio de nombres de `BPELUnit` y el resto, son prefijos propios de esta composición y el prefijo para los envoltorios SOAP.

A continuación encontramos la sección de despliegue (líneas 10 a 20). En ella podemos comprobar que la composición se comunica con dos *mockups*: el asesor y el aprobador.

Finalmente, de la línea 22 en adelante, aparece la sección de los casos de prueba en la que debe ir incluido cada bloque «testCase».

En concreto, se ha incluido un caso de prueba de ejemplo en el que el cliente solicita un préstamo por valor de 150.000 €(líneas 30 a 36), que es aceptado directamente por el aprobador (líneas 52 a 58), sin que tenga que intervenir el asesor (línea 63). El aprobador comunica al cliente su decisión (líneas 38 a 44).

1.5.3.3. WS-CDL

WS-CDL es un lenguaje utilizado para la definición de servicios dentro de la plataforma SOA (Service Oriented Architecture), basado en XML y cuyo objetivo es la descripción del comportamiento de cada uno de los servicios

establecidos para lograr un objetivo común.

Permite la descripción sin ambigüedades de las colaboraciones establecidas entre servicios, determinando un protocolo de negociación. De esta manera, cada organización puede desarrollar de manera independiente su propio papel en la colaboración siempre y cuando se respete el "contrato global" para que de esta manera se garantice la interoperabilidad.

1.5.3.4. WS-I Basic Profile

Dentro de la arquitectura SOA para la implementación de Servicios Web, la interoperabilidad es tal vez el principio más importante: debe ofrecer importantes beneficios de interoperabilidad y permitir la ejecución de WS distribuidos en múltiples plataformas de software y arquitecturas de hardware.

Mientras que el trabajo en el W3C se centra en las nuevas versiones de las especificaciones del núcleo de los Servicios Web, existe otra organización independiente que está prestando más atención a la interoperabilidad. Se trata de la WS-I (Web Services Interoperability Organization) y representa un esfuerzo de la industria en este sentido. Su objetivo es fomentar y promover la interoperabilidad de WS sobre cualquier plataforma, sobre aplicaciones y sobre lenguajes de programación. Pretende ser un integrador de estándares para contribuir al avance de los WS de una manera estructurada y coherente.

Hay varios estándares que necesitan ser coordinados para llevar a buen término las cuestiones de interoperabilidad de servicios. La WS-I ha organizado los estándares que afectan a la interoperabilidad de los WS en una pila basada en funcionalidades. Dichos estándares se están desarrollando en paralelo y de manera independiente. Para superar estas cuestiones, la WS-I ha desarrollado el concepto de "perfil" (profile). La WS-I define un perfil como:

"Un conjunto de definiciones/especificaciones comúnmente acep-

> tadas por la industria y a partir del apoyo a estándares basados en XML, junto con un conjunto de indicaciones que recomiendan cómo se deben usar las especificaciones para desarrollar servicios web interoperables entre sí."

Dado que dicha organización, de carácter abierto, está compuesta por las principales compañías de desarrollo de software, tales como IBM, Microsoft o Sun Microsystems, hay un claro compromiso por incluir todos estos estándares en el mundo de la programación actual y futura.

WS-I ha publicado varios perfiles hasta el momento. El primer perfil para la verificación de interoperabilidad fue precisamente, Basic Profile 1.1, el cual describe un conjunto de guías para el uso de un WS más allá de los protocolos básicos. Estas guías se basan en validaciones a nivel del esquema XML, de los mensajes y paquetes SOAP.

En concreto, la versión 1.1 del WS-I Basic Profile [63] guía el uso conjunto de SOAP 1.1, WSDL 1.1 y UDDI 2.0. Esta versión es la que se ha considerado que siguen las entradas de *ServiceAnalyzer*. Sin embargo, ya existen dos nuevas versiones: la 1.2 que ofrece soporte para SOAP 1.1, WSDL 1.1, UDDI 2.0, WS-Addressing [3] 1.0 y MTOM (Message Transmission Optimization Mechanism) [4] , y la 2.0, que guía el uso de SOAP 1.2, WSDL 1.1, UDDI 2.0, WS-Addressing y MTOM.

Existen otros perfiles que están diseñados para extender este primero, como por ejemplo Simple SOAP Basic Profile y Basic Security Profile. Simple SOAP Basic Profile 1.0 describe los elementos válidos dentro de los mensa-

[3] Web Services Addressing forma parte de la familia de especificaciones relacionadas con los WS desarrolladas por el W3C. Proporciona mecanismos neutrales, independientes de la capa de transporte, para direccionar mensajes y WS.

[4] MTOM es un mecanismo optimizado, también declarado por la W3C, para la transmisión de datos binarios a través de un WS. La eficiencia de este mecanismo reside en el tamaño del texto que genera al transformar los datos binarios a texto, codificados en base64.

jes SOAP, que son de importancia para el transporte y procesamiento de los mensajes. Basic Security Profile 1.0 es otro perfil que provee guías de implementación para estándares como WS-Security.

CAPÍTULO 2

Desarrollo del calendario

El proyecto se ha realizado a lo largo de unos doce meses, desde mayo de 2010 hasta mayo de 2011. No obstante, excepto en los primeros meses, no se ha dedicado todo el tiempo única y exclusivamente al desarrollo del proyecto, sino que se ha compaginado con otras tareas.

En realidad, la idea del proyecto surgió en octubre de 2009. A la alumna le fue concedida una beca de colaboración del MEC con el Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Cádiz. Debía participar en tareas de apoyo dentro del proyecto SOAQSIM (TIN-2007-67843-C06-04) [1] en el que trabaja el grupo de investigación, por entonces SPI&FM. En concreto, su colaboración facilitaría la aplicación de pruebas automáticas de software a composiciones de servicios a través del "Analizador de casos de prueba en WS-BPEL". Sin embargo, en ese mismo momento `BPELUnit` estaba en plena transformación, ya que se preparaba para la incorporación de Apache Velocity [21]. Con esta incorporación, `BPELUnit` podría construir el

[1] Proyecto financiado por el Programa Nacional de I+D+i del Ministerio de Educación y Ciencia y fondos FEDER en el que participó el grupo SPI&FM hasta el año 2010.

cuerpo de los mensajes SOAP usando plantillas. Las plantillas permitirían generar los mensajes a partir de una fuente de datos y una serie de variables predefinidas, lo que daría al usuario la posibilidad de definir diversos casos de pruebas, con la misma actividad, pero diferente contenido en los mensajes. Con este cambio, se daba un vuelco a la idea original del proyecto tal y como se había definido inicialmente, así que se decidió esperar a que `BPELUnit` tuviese la capacidad de usar plantillas para reformular el PFC.

2.1. Etapas

En general, se puede decir que se ha seguido un enfoque incremental para el desarrollo del sistema, con el que se obtuvieron unos requisitos iniciales que posteriormente se fueron *incrementando* hasta obtener el producto final. Por tanto, iterativamente se ha ido ampliando y mejorando el sistema según los resultados de las pruebas y de acuerdo con las indicaciones de los tutores del PFC, con los que se ha mantenido una comunicación continua.

De manera resumida, podemos decir que el proceso de desarrollo seguido comprende las siguientes tareas: la elicitación de requisitos, el estudio de las tecnologías, la lectura de interfaces WSDL, la creación de *WSDL2XSDTree*, el analizador de tipos, el generador de declaraciones, el generador de plantillas, la adaptación de *ServiceAnalyzer* para su uso a través de la línea de órdenes, la creación del paquete de instalación, la validación y mejora de la calidad del código y la documentación.

2.1.1. Elicitación de requisitos

Los requisitos de la aplicación fueron detallándose a través de una serie de reuniones con los miembros del grupo de investigación que trabajan más de cerca con `GAmera` y `BPELUnit`. Siguiendo el modelo iterativo, los requisitos

se han ido refinando a lo largo de todo el proyecto. Sin embargo, el primer mes se dedicó una especial atención a los mismos para saber a qué nos enfrentábamos.

2.1.2. Estudio de las tecnologías

Una vez obtenidos los requisitos del PFC se pasó a la fase de estudio de los lenguajes, herramientas y tecnologías que se utilizan en este PFC.

Cabe destacar el gran esfuerzo que se realizó durante esta fase, ya que, dado el contenido innovador de este PFC, las tecnologías estudiadas son todas tecnologías emergentes (algunas incluso se encuentran aún en fase experimental) para el desarrollo, prueba y explotación de sistemas informáticos.

La gran mayoría de las tecnologías usadas venían impuestas por las necesidades del grupo, sin embargo, en algunos casos, como por ejemplo el análisis de los componentes XML Schema, había un abanico de posibilidades, como se detallará más adelante. Por ello, en esta etapa se llevaron a cabo, asimismo, estudios comparativos de tecnologías para realizar la elección de la más adecuada para nuestro caso.

Entre las tecnologías estudiadas podemos destacar WSDL, SOAP, WS-BPEL, XSLT, XPath, XML Schema, `JUnit`, Apache Velocity, `XMLBeans`, WSDL4J, XSOM, `NetBeans` y `Maven`.

2.1.3. Lectura de las interfaces WSDL

Primeramente, se definió la estructura que contendría el catálogo de salida con las plantillas y las declaraciones. En el caso de las declaraciones, fue necesario determinar los tipos de datos que se iban a contemplar y las restricciones que se les podía asociar. Una vez hecho todo esto se creó un componente capaz de leer las interfaces WSDL de las composiciones de servicios y rellenar la

información contextual de los mensajes, esto es, servicio, puerto, operación y si se trata de una petición, una respuesta o un error.

2.1.4. WSDL2XSDTree

El paso previo a poder generar, tanto plantillas, como las declaraciones de tipo asociadas a las variables de las mismas, era poder leer los esquemas de una manera cómoda y organizada. Esto era necesario para que durante la lectura se tratara de forma uniforme tanto a los esquemas incrustados en ficheros WSDL, como a los se importan desde un fichero WSDL y están en un fichero XSD (XML Schema Definition) externo, como a los que se importan desde un fichero WSDL y están incrustados en otro fichero WSDL. Se puede decir que esto es lo que hace, *a grosso modo*, *WSDL2XSDTree*.

2.1.5. Analizador de tipos

Se diseñó la lógica necesaria para analizar un tipo o elemento XML Schema y traducirlo en un *árbol de sintaxis intermedia* (AST), esto es, una jerarquía de nodos, los cuales contienen la información necesaria para posteriormente poder generar las plantillas y las declaraciones correspondientes.

En sucesivas iteraciones se fue aumentando el número de restricciones XML Schema contempladas. Por otro lado, en un primer momento sólo se daba soporte a aquellos tipos que procedían un mensaje SOAP con estilo *document/literal*; ya en iteraciones posteriores se agregó la capacidad de construir el árbol cuando la codificación es *RPC/literal*.

2.1.6. Generador de declaraciones

Se implementaron las clases necesarias para construir a partir de un AST (Abstract Syntax Tree) las plantillas correspondiente. Tanto para este paso,

como para el siguiente, fue necesario el aprendizaje del patrón Visitante.

2.1.7. Generador de plantillas

Se implementaron las clases necesarias para generar las plantillas a partir de un AST.

2.1.8. Validación y mejora de la calidad del código

Esta fase comprende toda una serie de tareas dedicadas a mejorar la calidad del producto: modificaciones en el código siguiendo métricas de calidad, rediseño de clases y pruebas para favorecer la reutilización, inclusión de comentarios, etc.

2.1.9. Paquete de instalación y línea de órdenes

En esta etapa se hicieron los ajustes necesarios para que *ServiceAnalyzer* pueda ser fácilmente instalable en el equipo de un futuro usuario y pueda utilizarlo desde la línea de órdenes.

2.1.10. Documentación

La memoria del presente PFC se ha ido elaborando a lo largo del tiempo, a medida que avanzábamos en cada una de las etapas. No obstante, su redacción ha sido realizada mayormente en los dos últimos meses.

Cuando hablamos de documentación, no nos referimos únicamente a la presente memoria, sino también a los manuales de usuario y desarrollador.

2.2. Diagrama Gantt

Se ha elaborado un diagrama Gantt (Figura 2.1 en la página 41) para poder visualizar con mayor facilidad la distribución de las tareas.

Se ha considerado que un día ideal equivale a la cantidad de trabajo que puede realizarse en un día por una persona, sin distracción alguna y a máximo rendimiento.

Se ha empleado la herramienta `Gantt Project` para dibujar el diagrama y `GNOME Planner` para calcular las fechas con mejor precisión. Ambos son código abierto: la primera está hecha en Java y disponible en `http://ganttproject.sourceforge.net`, y la segunda se halla escrita en C++ con la biblioteca GTK+ y se puede encontrar bajo la dirección `http://live.gnome.org/Planner`.

2.3. Porcentajes de esfuerzo

En cuanto a la cantidad de esfuerzo dedicado a cada fase, se muestra en 2.1. Cabe destacar el hecho de que los porcentajes de esfuerzo dedicados a la fase de prueba de la aplicación y a su propia construcción sean prácticamente equivalentes. Esto se debe a que, como ya se ha comentado, se trata de un proyecto que tendrá continuidad en proyectos de investigación del grupo, por lo que se requiere una buena batería de pruebas, así como un diseño claro y consistente, que facilite la creación de nuevas versiones. Por otro lado, también ha influido la amplia variedad de entradas potenciales que puede recibir el programa debido a lo abierto que son los estándares con los que se trabaja (XML Schema, especialmente).

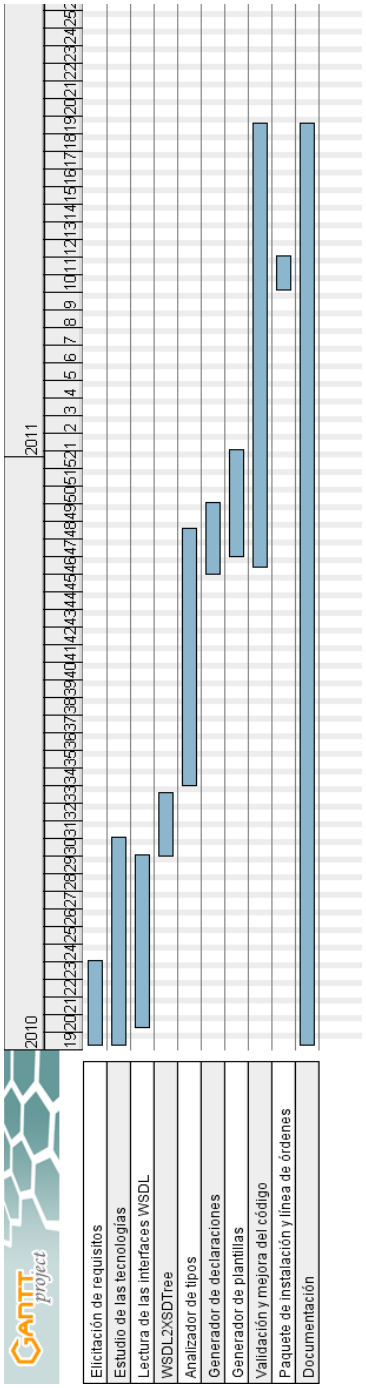

Figura 2.1: Diagrama de Gantt

Fase del proceso	Días ideales	Porcentaje
Análisis	5,6	6 %
Diseño	14,7	16 %
Construcción	36,8	40 %
Pruebas	34,9	38 %
Total	92	100 %

Tabla 2.1: Porcentajes de esfuerzo

CAPÍTULO 3

Descripción general del proyecto

3.1. Perspectiva del producto

3.1.1. Entorno de los productos

El proyecto está formado por la aplicación principal, *ServiceAnalyzer* y el proyecto *WSDL2XSDTree*, que es a su vez una de las dependencias del primero. Ambos productos se hallan escritos desde cero y tienen entidad propia. Sin embargo, en un futuro, *ServiceAnalyzer* será uno de los componentes del analizador para la generación de casos de prueba que complementará la herramienta de generación de mutantes `GAmera`, permitiendo mejorar la calidad del conjunto de casos de prueba inicial que se le suministra a la misma. El objetivo que se pretende alcanzar es que los nuevos casos de prueba permitan diferenciar entre mutantes equivalentes y mutantes resistentes, así como reducir el número de casos de prueba del conjunto.

3.1.1.1. GAmera

GAmera [14, 15] es la herramienta para la generación y ejecución automática de mutantes para composiciones de Servicios Web en WS-BPEL [44]. Incorpora un mecanismo de optimización que permite seleccionar un subconjunto de los mutantes totales que pueden generarse. Esto se logra mediante la utilización de un algoritmo genético que genera y selecciona sólo los mutantes de mayor calidad, reduciendo el coste computacional que implicaría la ejecución de todos los mutantes. Los resultados que proporciona esta herramienta permiten mejorar la calidad de los casos de prueba.

GAmera está constituida por tres componentes principales cuya interacción podemos observar en 3.1:

- **Analizador:** es el componente de la herramienta primero en actuar. Recibe como entrada la composición WS-BPEL a probar y determina los operadores de mutación que se le pueden aplicar. Los operadores se muestran en pestañas independientes, según la categoría del operador, permitiéndose al usuario incluso poder determinar el conjunto de operadores a utilizar de los que son aplicables a la composición WS-BPEL con la que se está trabajando.

- **Generador de mutantes:** es el siguiente componente que entra en acción, partiendo de la información que se recibe del analizador. La herramienta nos da la posibilidad de generar todos los mutantes posibles, o bien un subconjunto de éstos que va a ser seleccionado por el ya citado algoritmo genético.

  En este último caso, se llamará al componente denominado *"Generador genético de mutantes"*, que está compuesto por dos elementos a su vez. El primero es el algoritmo genético en sí, denominado *"Búsqueda genética*

de mutantes", en el que cada individuo representa a un mutante, capaz de generar y seleccionar de forma automática un conjunto de mutantes. Esta selección se realiza aplicando una función de aptitud que mide su calidad en función de si hay o no casos de prueba que lo matan. El segundo elemento es el *"Conversor"*, que transforma un individuo del algoritmo genético en un mutante WS-BPEL. Para realizar esta conversión, se utilizan hojas de estilos XSLT, una por cada operador de mutación.

- **Sistema de ejecución:** a medida que se van generando los mutantes, este tercer componente los ejecuta frente a un conjunto de casos de prueba, distinguiéndose tres posibles estados para cada mutante según la salida que producen:

  Muerto La salida del mutante es diferente a la del proceso original para al menos un caso de prueba.

  Vivo La salida del mutante es la misma que la del proceso original para todos los casos de pruebas suministrados.

  Equivalente La salida del mutante y la del programa original es siempre la misma para todos los casos de prueba.

  Resistente El conjunto de casos de prueba no es suficiente para detectarlo.

  Erróneo Se ha producido un error en el despliegue del mutante y no se ha podido ejecutar. La existencia de este estado permite determinar si el diseño e implementación de los operadores de mutación es adecuado, o bien, si se están generando mutantes que no se pueden desplegar.

  Para la ejecución del programa original y los mutantes, `GAmera` emplea el motor WS-BPEL 2.0 ActiveBPEL 4.1 [2] y `BPELUnit` [41], una biblio-

teca de pruebas unitarias para WS-BPEL que utiliza ficheros XML para describir los casos de prueba.

GAmera permite visualizar los resultados obtenidos en la ejecución de los mutantes. Muestra el número total de mutantes generados, el de mutantes muertos, vivos y erróneos. También muestra estos valores para cada operador de mutación utilizado. A partir de estos valores se puede medir la calidad del conjunto de casos de prueba utilizado.

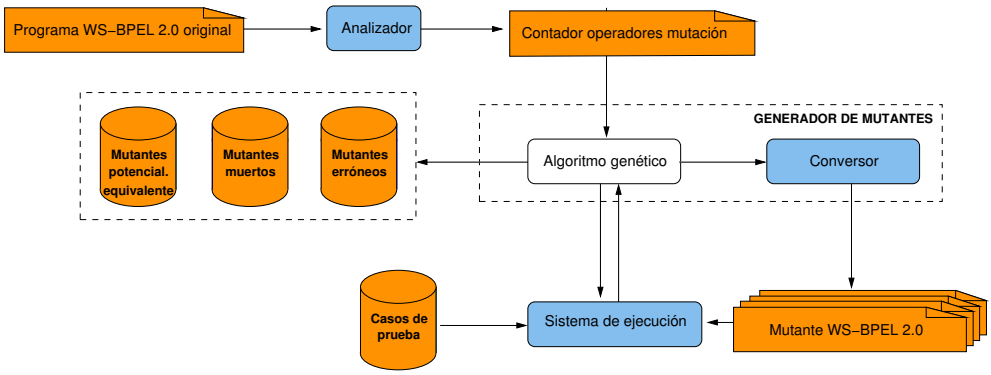

Figura 3.1: Componentes de GAmera

3.1.1.2. Analizador para la generación de casos de prueba

El analizador para la generación de casos de prueba en WS-BPEL de GAmera estará constituido por al menos tres componentes:

1. **BPELUnit**: (véase [41]) biblioteca de prueba unitaria WS-BPEL [44]. Un aspecto a destacar es que, desde la reciente incorporación del lenguaje de plantillas Apache Velocity, se pueden utilizar plantillas en los ficheros BPTS (ver §1.5.3.2). Éstas permitirán generar los mensajes a partir de una fuente de datos y una serie de variables predefinidas.

2. **El algoritmo ocupado de generar los elementos de la fuente de datos** que usará `BPELUnit`. Necesitará saber cuáles son los tipos de los elementos de la fuente de datos para trabajar: si son enteros, cadenas, listas de enteros, listas de cadenas, listas anidadas, etc. También tiene que saber cuáles son los valores válidos que puede tomar un elemento de la fuente de datos: XML Schema permite poner restricciones como "mayor de 2" o "máximo 3 caracteres".

3. **ServiceAnalyzer:** con sólo los dos componentes anteriores podría ser suficiente para investigar acerca de la generación de casos de prueba. Tendrían que generarse manualmente las plantillas para cada uno de los mensajes y las declaraciones de tipos y listo. Sin embargo, habría que tener cuidado con las restricciones que imponen WSDL, SOAP, XML Schema y el WS-I Basic Profile 1.1, que hacen que el proceso sea incómodo y propenso a errores difíciles de depurar. Y ahí es donde entra este Proyecto. *ServiceAnalyzer* automatizará el trabajo de generar las plantillas que necesita `BPELUnit` y las declaraciones de tipos que necesita el algoritmo. Tras su ejecución el usuario tendrá la plantilla y las declaraciones para cada mensaje de cada servicio de un WSDL, de acuerdo al *binding* que use: su entrada, su salida, o alguno de sus fallos. De este modo, se libera al autor del BPTS de estar pendiente de todas esas restricciones y únicamente se ocupará de reunir las plantillas deseadas en el fichero de pruebas.

Listado 3.1: Estructura de un fichero *.bpts*

```
1 <tes:testCases>
2     <tes:testCase ...>
3       <tes:setUp>
4         <tes:script>...</tes:script>
```

```
5           <tes:dataSource type="velocity">
6             <tes:property name="iteratedVars">
7               variable_1 variable_2 ... variable_m
8             </tes:property>
9             <tes:contents>
10  #set($variable_1 = [valor_11, valor_12,..., valor_1n])
11  #set($variable_2 = [valor_21, valor_22,..., valor_2n])
12  ...
13  #set($variable_m = [valor21, valor_22,..., valor_2n])
14            </tes:contents>
15          </tes:dataSource>
16        </tes:setUp>
17
18        <tes:clientTrack>
19          <tes:sendReceive
20              service="..."
21              port="..."
22              operation="...">
23            <tes:send fault="false">
24              <tes:template>
25                <ns:mensaje_1 ns="...">
26                  <ns:parteVariable_1>$variable_1</ns:parteVariable_1>
27                </ns:mensaje_1>
28              </tes:template>
29            </tes:send>
30          ...
```

Para tener una idea más concreta, con respecto a `BPELUnit`, si expandimos la sección de casos de prueba (ver figura 1.2 de §1.5.3.2) de un fichero BPTS que utilice plantillas Velocity, podríamos encontrar un fragmento de código similar al del listado 3.1.

Las plantillas se incluyen en los elementos «template» y son construcciones que representan los mensajes a enviar por el cliente (como en 3.1) o por

un *mockup*. Si se parametrizan con el uso de variables, permiten reutilizar un mismo caso de prueba («testCase») para probar el mismo mensaje con diferentes valores.

Estas plantillas son las que genera *ServiceAnalyzer*. Estarán clasificadas por servicio, puerto, operación y tipo de mensaje (entrada, salida o fallo) para saber en qué lugar del BPTS debe incrustarse.

En el elemento «setUp» se encuentra la fuente de datos. Es donde se declaran los nombres de las variables que serán utilizados en las plantillas y se asigna a cada una de ellas una lista de valores. El primer valor de la lista es el que tomará la variable en la plantilla en la primera ejecución (primer caso de prueba), el segundo, el valor correspondiente al segundo caso de prueba y así sucesivamente.

Las declaraciones que genera *ServiceAnalyzer* serán empleadas por el algoritmo descrito en el segundo punto para generar los valores de la fuente de datos.

El listado es tan sólo un ejemplo, para conocer con más detalle la estructura que puede tener un fichero BPTS que utilice plantillas Apache Velocity, véase el anexo B.

3.1.2. Interfaces software y hardware

La salida de *ServiceAnalyzer* es el catálogo de plantillas y declaraciones en formato XML, por lo que la interfaz entre éste y cualquier sistema que haga uso de su salida está bien definida a través de su XSD, que permite conocer el formato de la salida sin tener que examinar el programa.

En cuanto a la salida de *WSDL2XSDTree*, es una serie de uno o más ficheros XSD. Si el fichero proporcionado como entrada es `mi.wsdl` la raíz del árbol generado como salida será `mi.wsdl.xsd`. Dependiendo de la entrada

podrían crearse más ficheros nombrados como `mi.wsdl_1.xsd`, `mi.wsdl_2.xsd`, ..., `mi.wsdl_n.xsd`. En conclusión, cualquier sistema con capacidad para tratar ficheros XSD podrá tomar la salida de *WSDL2XSDTree*.

3.1.3. Interfaz de usuario

La herramienta *ServiceAnalyzer* no presenta una interfaz gráfica. Para interactuar con esta herramienta se utilizará una terminal.

3.2. Funciones

3.2.1. WSDL2XSDTree

Las funciones de *WSDL2XSDTree* son las siguientes:

- Extraer los XML Schema de un fichero WSDL dado, tanto los incluidos (incrustados o importados) en la sección «types» del mismo, como los derivados de los ficheros WSDL que importe el WSDL original de entrada.
- Organizar jerárquicamente los esquemas extraídos generando los ficheros XSD correspondientes que conformarán el árbol de salida.

3.2.2. ServiceAnalyzer

A continuación, se detallan las funciones de *ServiceAnalyzer*:

- Analizar las interfaces de una composición de Servicios Web descritas en WSDL 1.1.
- Generar las plantillas de todos los posibles mensajes SOAP que se intercambian entre la composición y los *mockups* descritos por los WSDL. Las

plantillas han de generarse de forma paramétrica, es decir, permitiendo que una misma plantilla sirva para construir mensajes con el mismo formato pero distinto contenido a través del uso de variables.

- Traducir los tipos que empleen los mensajes al sistema de tipos simplificado definido.
- Generar las declaraciones de los tipos de las variables empleadas en las plantillas.
- Reunir toda la información en el fichero catálogo de salida con formato XML.

3.3. Características del usuario

Como ya se ha comentado en más de una ocasión, uno de los objetivos de este PFC es completar la herramienta de generación de mutantes `GAmera`. Servirá de base para la implementación de un generador que pretende mejorar la calidad del conjunto de casos de prueba inicial que se le suministra a la herramienta. Sin embargo, el uso de *ServiceAnalyzer* no tiene por qué estar ligado a `GAmera`. En consecuencia, podemos distinguir dos tipos de usos y por tanto, dos tipos de usuarios para *ServiceAnalyzer*:

- Si *ServiceAnalyzer* es utilizado fuera del entorno de `GAmera`, el usuario deberá poseer conocimientos en Servicios Web, WSDL y SOAP.
- Si este producto se utiliza integrado en `GAmera`, además se requieren otros conocimientos. Como se ha visto, dicha herramienta se divide en dos partes: un algoritmo genético que dirige todo el proceso y un entorno que compara los mutantes con el proceso original, para comprobar si su comportamiento varía o no. Por tanto, los usuarios de este segundo

grupo, deberán tener conocimientos, al menos, de las pruebas de mutaciones de programas [17], del análisis de mutaciones y del lenguaje WS-BPEL 2.0 [44].

En ambos casos el perfil de usuario es experto, aunque más especifico en el segundo. Cabe destacar, que puesto que en la actualidad el grupo UCASE está creciendo en número de profesores integrantes y de alumnos que deciden colaborar en alguna de sus líneas de investigación, creemos que va a ser de gran ayuda para los mismos. Con los catálogos generados podrán estudiar de un modo más claro las composiciones de servicios, y por supuesto, construir con gran facilidad ficheros BPTS de casos pruebas, sacando partido a todas las novedades de `BPELUnit`.

Por último, *WSDL2XSDTree* también puede ser utilizado de modo independiente. Esta utilidad podrá utilizarla cualquier usuario con conocimientos básicos en XML Schema, ni siquiera es necesario que sepa WSDL, aunque sea éste el formato de entrada. Podría darse el caso, aunque *a priori* suene raro, de una persona que no sabe nada de WSDL, sin embargo dispone de un fichero WSDL y necesita utilizar los tipos definidos en él con un determinado fin.

3.4. Restricciones generales

3.4.1. Control de versiones

Cuando se construye software, los cambios son inevitables. Constantemente, se están creando nuevas versiones de un producto software.

No nos cansamos de repetir que este Proyecto tendrá continuidad en proyectos de investigación del grupo UCASE. Además, como ya se ha citado, se ha realizado un proceso incremental de desarrollo de software. Por todo esto,

es esencial llevar una buena Gestión de la Configuración del Software (SCM). Según la IEEE (Institute of Electrical and Electronics Engineers):

> "SCM es una disciplina formal de ingeniería que proporciona un conjunto de métodos y herramientas para identificar y controlar el software durante su desarrollo y utilización." [1]

Para este cometido, nos hemos apoyado en un sistema de control de versiones de todas las fuentes del proyecto.

Estos sistemas permiten almacenar todas las versiones de un árbol de ficheros, pudiendo así manipular todas las revisiones de cualquier fichero en cualquier momento.

Además de servir como una medida de seguridad contra la pérdida de información accidental, agilizan los cambios drásticos, ya que no hay que establecer medidas especiales por si fallaran: se pueden revertir los cambios hechos en cualquier momento.

En particular, se ha utilizado `Subversion`, un sistema que trata de resolver las insuficiencias del conocido CVS (Concurrent Versions System), pudiendo mantener revisiones de directorios completos, establecer propiedades especiales sobre los elementos del repositorio y enviar nuevas revisiones de forma atómica, entre otras cosas.

Cabe resaltar el libro [7] sobre `Subversion` que ha sido de gran ayuda a la hora de manejar dicha herramienta.

3.4.2. Lenguajes de programación y tecnologías

Los lenguajes de programación y las tecnologías que se han utilizado durante el desarrollo de este PFC son:

- Java: es el lenguaje en el que se han implementado los productos de este Proyecto.
- WS-BPEL: es el lenguaje en el que ha centrado su estudio el grupo UCA-SE para llevar a cabo la composición de Servicios Web.
- WSDL 1.1: es el lenguaje para describir las interfaces de Servicios Web que recibe como entrada *ServiceAnalyzer*.
- XML Schema: describe la estructura y las restricciones de los contenidos de los documentos XML. Se ha empleado para fijar la estructura del fichero catálogo de salida de *ServiceAnalyzer*, así como para la traducción de tipos usados en los mensajes, ya que los tipos a traducir se leen de los XML Schema que contienen los documentos WSDL de entrada.
- Apache Velocity: es un motor de plantillas basado en Java. Se puede utilizar para generar cualquier contenido mediante la creación de plantillas. En el caso de este PFC, se utiliza para generar datos estructurados en XML.
- `BPELUnit`: framework para realizar pruebas de unidad de composiciones WS-BPEL. Lo utilizamos indirectamente a través de `MuBPEL`.
- `MuBPEL`: este es el componente específico para el manejo de mutantes WS-BPEL en `GAmera`. Se ha utilizado para ejecutar contra las composiciones de servicios casos de prueba construidos a partir de los catálogos generados por *ServiceAnalyzer* para las mismas y comprobar que toda va bien.
- `XMLEye`: es un visor de XML reconfigurable mediante hojas de estilo XSLT (eXtensible Stylesheet Language Transformations) y preprocesadores definidos por el usuario. En concreto se ha empleado una distri-

bución con soporte para visualizar *logs* de la ejecución de un conjunto de casos de prueba de `BPELUnit`, es decir, la salida de `MuBPEL`.

El resto de tecnologías utilizadas, que se refieren a un nivel de implementación más concreto, se discutirán en el siguiente capítulo.

3.4.3. Sistemas operativos y hardware

Este Proyecto ha sido desarrollado en GNU/Linux, ejecutando la versión 9.10 (Karmic Koala) de la distribución Ubuntu, basada en Debian. Únicamente ha sido probado bajo esta versión y la versión 11.0 de la distribución openSUSE.

Al haber sido implementados en Java, tanto *WSDL2XSDTree* como *ServiceAnalyzer* pueden funcionar sobre cualquier entorno Java que implemente la J2SE (Java 2 Standard Edition) 5.0 como mínimo. Esto incluye evidentemente a los JRE 5.0 y 6.0 de Sun y a las versiones 6.0 y 7.0 de OpenJDK [1].

3.4.4. Bibliotecas y módulos usados

Para desarrollar tanto *ServiceAnalyzer* como *WSDL2XSDTree*, se han usado las siguientes herramientas:

NetBeans 6.9.1 [9] IDE (Integrated Development Environment) desarrollado por Sun Microsystems (ahora Oracle) que es, junto con `Eclipse`, uno de los más usados hoy en día. Es multiplataforma: se puede utilizar en Windows, Linux, Mac OS X y Solaris. `NetBeans IDE` es un producto libre y gratuito sin restricciones de uso. El código fuente está disponible

[1] OpenJDK es una iniciativa liderada por Sun que, en combinación con los esfuerzos del proyecto IcedTea, ha conseguido una versión prácticamente 100 % funcional y basada en software libre de las J2SE 6.0 y la futura 7.0.

para su reutilización de acuerdo con las licencias CDDL (Common Development and Distribution License) v1.0 y GPL (GNU General Public License) v2.

Aunque es particularmente popular en la comunidad Java, lenguaje en el que está implementado fundamentalmente, existen versiones para otros lenguajes, como C o C++. Dispone de herramientas para hacer rápidamente refactorizaciones en el código, facilitando enormemente cualquier cambio al diseño de una aplicación. Existe además un número importante de módulos [2] para extender el IDE.

JUnit 4.8.1 Popular marco de desarrollo de pruebas de unidad para aplicaciones Java, creado por Erich Gamma y Kent Beck (véase [26, 36]). Permite realizar la ejecución de clases Java de manera controlada y evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Está inspirado en el marco SUnit para Smalltalk, que ha originado otros marcos parecidos, como CppUnit para C++. Un libro excelente para introducirse en el mundo de las pruebas unitarias y el uso de `JUnit` es [30].

3.4.5. Herramientas de integración continua

3.4.5.1. Introducción a la integración continua

La integración continua (*Continuous Integration* o CI en inglés) es un conjunto de prácticas de ingeniería del software, propuesta inicialmente por Mar-

[2] La plataforma `NetBeans` permite que las aplicaciones sean desarrolladas a partir de un conjunto de componentes de software llamados *módulos*. Un módulo es un archivo Java que contiene clases de java escritas para interactuar con las API de `NetBeans` y un archivo especial (*manifest file*) que lo identifica como módulo. Las aplicaciones construidas a partir de módulos pueden ser extendidas agregándole nuevos módulos. Debido a que los módulos pueden ser desarrollados independientemente, las aplicaciones basadas en la plataforma `NetBeans` pueden ser extendidas fácilmente por otros desarrolladores de software.

tin Fowler [11, 25], que aumentan la velocidad de entrega de software disminuyendo los tiempos de *integración*. Se entiende por integración la compilación y ejecución de pruebas en todo un proyecto.

La integración continua es un proceso que permite comprobar *continuamente* que todos los cambios que lleva cada uno de los desarrolladores no producen problemas de integración con el código del resto del equipo. Los entornos de integración continua construyen el software desde el repositorio de fuentes y lo despliegan en un entorno de integración sobre el que realizar pruebas.

Cuando un desarrollador va a realizar alguna modificación en el código, lo primero que hace es tomar una copia del código actual para trabajar con ella en su directorio local de trabajo. Pero puede haber al mismo tiempo otros desarrolladores haciendo cambios y que alguno de ellos o varios se adelanten a subir al repositorio sus modificaciones. La copia con la que trabaja el desarrollador original deja de reflejar el código del repositorio. Cuando éste sube su código, primeramente debe actualizarlo para reflejar los cambios que se han producido en el repositorio desde que hizo su copia. Cuantos más cambios haya en el repositorio, más trabajo debe hacer el desarrollador antes de subir los suyos.

Pero hay que tener en cuenta que el repositorio se puede convertir en algo tan diferente a la línea base del desarrollador que puede caer en lo que se suele llamar “infierno de integración”, donde el tiempo que usan para integrar es mayor que el tiempo que le han llevado sus cambios originales. En el peor de los casos, los cambios que está haciendo el desarrollador podrían tener que ser descartados y el trabajo debería rehacerse.

Para configurar un entorno de integración continua debemos seguir una serie de directrices o buenas prácticas, que son las siguientes:

- Mantener un repositorio de código. Esta práctica recomienda el uso de un sistema de control de versiones para el código fuente del proyecto. Todos los artefactos que son necesarios para construir el proyecto se colocan en el repositorio.

- Automatizar la construcción. El sistema debería ser construible usando una única orden. La automatización de la construcción debería incluir la integración, que con frecuencia conlleva un despliegue en un entorno similar al de producción. En muchos casos, el script de construcción no sólo compila binarios, sino que también genera documentación, páginas web, estadísticas y distribución.

- Hacer las pruebas propias de la construcción. Esto toca otro aspecto de mejor práctica, desarrollo orientado a pruebas. Esta es la práctica de escribir una prueba que demuestre la falta de funcionalidad en el sistema y entonces escribir el código que hace que esa prueba se pueda pasar. Una vez que el código está construido, se deben pasar todas las pruebas para confirmar que se comporta como el desarrollador esperaba que lo hiciera.

- Mantener la construcción rápida. La construcción ha de ser rápida, de tal manera que si hay un problema con la integración, se identifica rápidamente.

- Probar en un clon del entorno de producción. Tener un entorno de pruebas puede conducir a fallos en los sistemas probados cuando se despliegan en el entorno de producción, porque el entorno de producción puede diferir del entorno de pruebas en una forma significativa. Esto evita problemas que todos hemos vivido del tipo “Pues a mí en local me compila . . . ”.

- Facilitar conseguir los últimos entregables. Mantener disponibles las construcciones para las personas involucradas en el proyecto puede reducir la cantidad de re-trabajo necesaria cuando se reconstruye una característica que no cumplía los requisitos. Adicionalmente, las pruebas tempranas reducen las opciones de que los defectos sobrevivan hasta el despliegue. Encontrar incidencias también de forma temprana, en algunos casos, reduce la cantidad de trabajo necesario para resolverlas.

- Todo el mundo puede ver los resultados de la última construcción. La comunicación es un elemento esencial en esta metodología y por ellos, todos los implicados deben poder ver a cada momento el estado actual del sistema y los últimos cambios realizados.

- Despliegue automático. Puesto que en la integración, intervienen diversos entornos, es importante tener un script que automatice el despliegue en cualquiera de ellos.

La integración continua en sí misma hace referencia a la práctica de la integración *frecuente* del código de uno con el código que se va a liberar. El término frecuente es abierto a interpretación, pero a menudo se interpreta como “muchas veces al día”. Las grandes ventajas de la integración continua se derivan del hecho de probar nuestro código “en tiempo real”. Esto está dentro del concepto *Kaizen* o de mejora continua, cuyas ventajas pueden resumirse a la reducción de tiempo de integración con una fiabilidad enorme. De un modo más detallado, las ventajas de usar esta metodología son:

- Se reduce el tiempo de integración gracias a su automatización.

- Posibilita la detección temprana de errores (código no operativo/incompatible, cambios conflictivos), minimizando los costes de resolución. Los

problemas de integración se detectan y arreglan continuamente, no son hábitos de último minuto antes de la fecha de liberación.

- Cuando las pruebas unitarias fallan, o se descubre un defecto, los desarrolladores pueden revertir el código base a un estado libre de defectos, sin perder tiempo depurando.
- Se realizan pruebas unitarias inmediatas de todos los cambios.
- Se dispone constantemente de una construcción actual para propósitos de pruebas, demo o liberación.
- El impacto inmediato de subir código incompleto o no operativo actúa como incentivo para los desarrolladores para aprender a trabajar de forma más incremental con ciclos de retroalimentación más cortos.

No obstante, la integración continua, como todo, tiene sus inconvenientes: se hace necesario un servidor dedicado a la construcción del software, hay que controlar la sobrecarga por el mantenimiento del sistema y además, el impacto inmediato al subir código erróneo provoca que los desarrolladores no hagan tantos *commits* como sería conveniente como copia de seguridad.

3.4.5.2. Implantación

El grupo UCASE con el que se ha colaborado decidió montar un sistema de integración continua para albergar los proyectos creados por el mismo. Los motivos de esta decisión fueron varios:

- La necesidad de una buena estructura para el código y esquemas de compilación e instalación mejores que los esquemas hechos a mano (tarea a tarea) con Ant, que hacía que cada proyecto fuese su propio mundo para compilar.

- Se tenían proyectos que no compilaban a menos que se preparara un entorno específico.

- Muchos de los proyectos están relacionados entre sí, tienen dependencias comunes, etc.

- A veces el código de proyectos en Java no daba problemas en la máquina del desarrollador original, pero dejaba de compilar o no superaba las pruebas en otra diferente.

El servidor de integración es un servidor virtualizado que se ejecuta dentro de Teseo. Actualmente, está limitado a 2GiB de RAM y 2 CPU, para que no pueda absorber todos los recursos si algo va mal durante unas pruebas unitarias. Se accede a través de Neptuno [28], que hace de *proxy* inverso.

El esquema básico montado consta de la interacción de cinco herramientas de distribución libre: `Jenkins`, `Subversion` [3], `Maven`, `Nexus` y `Sonar`. A continuación, describimos cada una de ellas:

Jenkins [37] Versión de la herramienta `Hudson` creada por la comunidad de software libre tras disputas con Oracle acerca del control de la marca registrada y la infraestructura del proyecto. `Hudson` es una herramienta de integración continua de código abierto, creada por Kôsuke Kawaguchi (empleado de Sun) en su tiempo libre, que se encarga de monitorear la ejecución de tareas repetidas, tales como la creación de un proyecto o la ejecución de tareas con un *cron* [4]. Es muy similar a otras herra-

[3] También admite `Git`, pero durante la elaboración del presente PFC se ha utilizado `Subversion`.

[4] *Cron* es el nombre del programa que permite a usuarios Linux/Unix ejecutar automáticamente órdenes o scripts a una hora o fecha específica. En las páginas del manual de cron se define como un demonio que ejecuta programas agendados. Su uso suele estar ligado a tareas administrativas.

mientas como `Continuum` y `Cruise Control`, aunque destaca por las siguientes características:

- Es fácil de instalar, ya que se distribuye como un fichero `war`.
- Todas las tareas de administración se realizan usando una interfaz web, lo cual facilita enormemente su configuración.
- Soporta notificación vía IM (Instant Messaging), e-mail y RSS (Really Simple Syndication).
- Genera gran cantidad de informes para `JUnit` y `TestNG`.
- La herramienta puede extenderse y personalizarse fácilmente mediante *plugins*.
- Soporta CVS y `Subversion` para el control de cambios.

En [57] podemos encontrar una tabla en la que se pueden observar las ventajas del predecesor de `Jenkins`, `Hudson`, frente a otras herramientas de CI existentes.

Subversion Sistema de control de versiones ya descrito en §3.4.1.

Maven 3.0 [22, 32] Herramienta para automatizar la gestión de software escrito en Java, incluyendo compilación, pruebas y despliegue, entre otras tareas. Fue creada dentro del proyecto Jakarta (2002), aunque actualmente el proyecto pertenece a la *Apache Software Foundation*.

Posee una funcionalidad similar a la de `Apache Ant`. La diferencia principal entre ambas es que en `Ant` las acciones a realizar se definen en forma procedural paso por paso, mientras que con `Maven` se declara qué *plugins* se van a utilizar, con qué configuración y con qué dependencias y `Maven` se encarga del orden en el que se utilizan las cosas para lograr el objetivo declarado.

Los objetivos (*goals*) de `Maven` son las unidades mínimas de ejecución de las que disponemos durante su uso (ver figura 3.2). Un grupo de objetivos conforman un *plugin*. La ejecución de un *goal* se dispara desde línea de comandos invocando `Maven` con el nombre del *plugin* que lo contiene. Los objetivos `Maven` más usados se recogen en la tabla 3.1.

Objetivo	Función
clean	Limpia
compile	Compila
test	Compila y prueba
package	Compila, prueba y empaqueta
site	Crea web en `target/site/index.html`

Tabla 3.1: Objetivos básicos de Maven

`Maven` utiliza un POM (Project Object Model) para describir el proyecto software a construir, sus dependencias con otros módulos y componentes externos, así como el orden de construcción de los elementos. Es un archivo basado en un formato XML que se ubica en la raíz del proyecto o módulo. Viene con objetivos predefinidos para realizar ciertas tareas tales como la compilación del código y su empaquetado.

Una característica clave de `Maven` es que está listo para usar en red. Nos permite publicar fácilmente binarios que incluyen todas las dependencias, de forma que sea simplemente descargar y utilizar. Si se desea utilizar una nueva biblioteca, por ejemplo, sólo tendrá que añadirse a las dependencias (al estilo de un paquete Debian) y dejar que `Maven` las descargue cuando haga falta.

El funcionamiento de `Maven` se basa en el uso de un repositorio a donde ir a buscar las dependencias. Cuando `Maven` sale a buscar y consigue una dependencia la guarda en el repositorio local que es un directorio en la máquina del usuario (`~/.m2/repository`). Las siguientes veces

que necesite esta dependencia la obtendrá del directorio local, haciéndolo mucho más rápido que la primera vez.

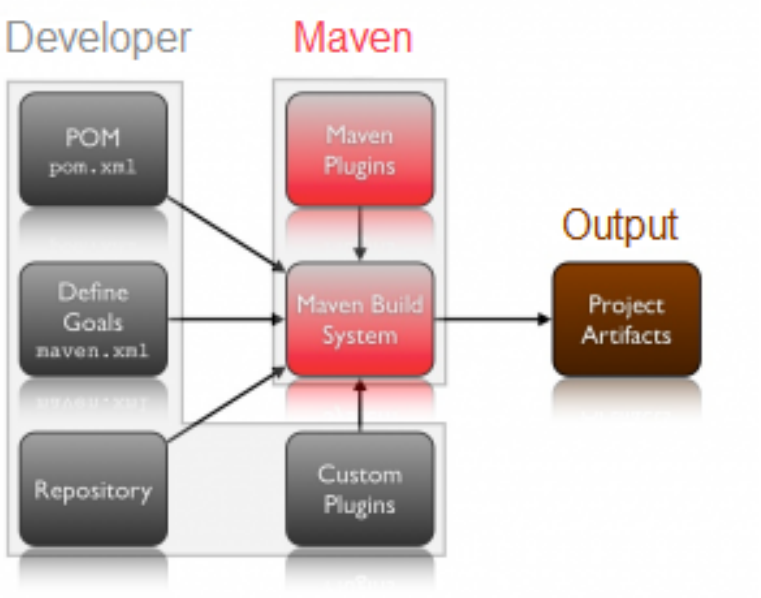

Figura 3.2: Arquitectura de Maven

Opcionalmente, se puede utilizar también un tercer repositorio, que haga de *proxy* para mejorar el rendimiento y permitir tener un mayor control de las versiones de dependencias que utilizarán un grupo de desarrolladores. En el grupo disponemos de uno: nuestra instancia `Nexus`.

`Maven` utiliza el término artefacto (*artifact*) para denominar a la unidad mínima que maneja en su repositorio. Puede ser por ejemplo un `jar`, un `ear`, un `zip`, etc. Cuando `Maven` compila y empaqueta código, produce también artefactos que instala en el repositorio.

Otra gran ventaja de `Maven` es que ayuda a imponer que los proyectos sigan una estructura uniforme. Las opciones de compilación, empaquetado, etc. pueden hacerse comunes a todos los proyectos, dejando una configuración mucho más consistente e independiente de los detalles de

cada proyecto. Maven también puede generar proyectos para Eclipse y NetBeans, con lo que sólo tenemos que mantener un sistema.

Otro aspecto interesante de Maven es que, a partir de la información del POM y del código fuente, proporciona al usuario información de los proyectos que puede llegar a serle muy útil: árboles de dependencias, listas de direcciones, informes de las pruebas, referencias cruzadas entre las fuentes, etc. Asimismo, proporciona guías de buenas prácticas para el desarrollador.

Nexus La distribución de Maven por defecto se descarga los artefactos del repositorio principal de Maven. Si bien esto a nivel personal es factible, cuando varios desarrolladores se tienen que descargar los mismos jars pesados una y otra vez es un despilfarro de ancho de banda y de tiempo considerable. Por otro lado, a una organización podría interesarle controlar o restringir de algún modo los artefactos que pueden descargarse los desarrolladores (para que descarguen una misma versión de una biblioteca, para que no usen el repositorio con fines ajenos a la organización, para que sólo empleen dependencias con una licencia compatible a la del proyecto en el que trabajan, etc.). Por ello se suele instalar un gestor repositorio propio, siendo Nexus una de las mejores opciones. En [53] se enumeran los motivos por los que debe elegirse Nexus, entre ellos podemos destacar:

- Puesto que está creado por desarrolladores de Maven, la compatibilidad con esta herramienta y la eficiencia en las comunicaciones con su repositorio central son las máximas posibles.
- Es pionero en el formato de repositorio índice.
- Su configuración y mantenimiento son sencillos. Sin embargo, pue-

de ser usado de manera profesional y permite su extensión mediante *plugins*.

- Destaca por poseer el modelo de seguridad más robusto y configurable de entre los gestores de repositorios actuales.
- `Nexus` usa Apache Lucene para indexar y buscar en tiempo real, sin necesidad de tener repositorios de contenido o bases de datos.

Sonar Proyecto que se autodenomina como *"una plataforma para la administración de la calidad del código"* (ver [52]). Se trata de una herramienta que permite automatizar el análisis del código para gestionar aspectos tales como las nomenclaturas requeridas por una arquitectura o una metodología, el uso de buenas prácticas de programación, la repetición de código, el porcentaje de código cubierto por pruebas, ciertos parámetros de complejidad de clases y métodos, el porcentaje de código comentado, la evolución de las métricas a lo largo del tiempo, etc. En concreto, la plataforma `Sonar` permite gestionar la calidad del código controlando los siete ejes principales de dicha calidad del código:

1. Arquitectura y diseño
2. Duplicaciones
3. Pruebas unitarias
4. Complejidad
5. Errores potenciales
6. Reglas de codificación
7. Comentarios

`Sonar` realiza diversos análisis de nuestro código a través de herramientas tales como `Findbugs` [47], `Checkstyle` [56], `PMD` [8] o

`Cobertura` [13]. La información generada por estas otras herramientas se presenta de manera unificada a través de su interfaz en forma de métricas.

La interacción entre estas herramientas se muestra en la figura 3.3.

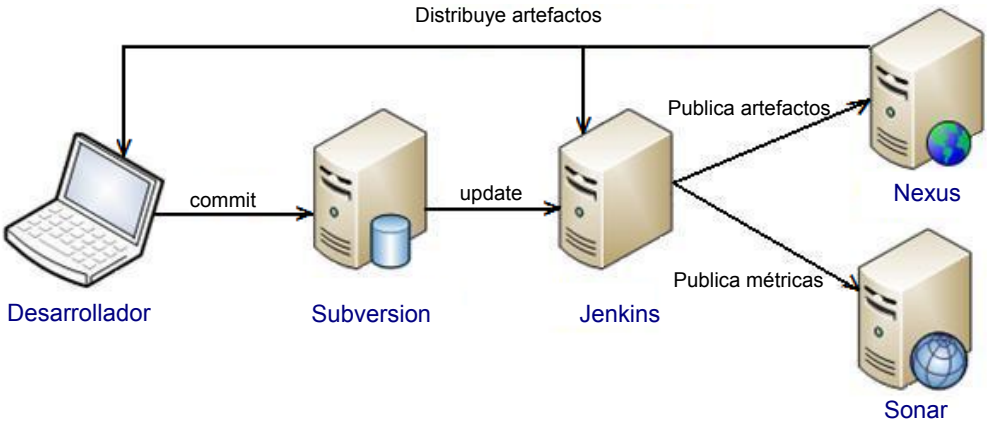

Figura 3.3: Sistema de integración continua

El proceso seguido es el siguiente:

1. El equipo de desarrollo realiza cambios al código fuente y realiza un *commit* en el sistema de control de versiones (`Subversion` o `Git`).
2. `Jenkins` detecta un cambio en el código y dispara la ejecución del *build*.
3. A través de `Maven`, se identifican los artefactos necesarios para su compilación.
4. `Nexus` provee las dependencias solicitadas.
5. Se inicia la compilación del código fuente.
6. Se ejecutan las pruebas unitarias utilizando `JUnit`.

7. `Sonar` pasa las pruebas de cobertura al código y genera los informes de métricas de calidad del código.

8. Se despliega el software en el servidor de aplicaciones y si se indica, se publican los nuevos artefactos a `Nexus`.

9. Si el proceso ocurrió con fallos, se procede a corregir. En caso contrario, se continua con el desarrollo.

3.5. Requisitos para futuras versiones

ServiceAnalyzer y *WSDL2XSDTree* se seguirán mejorando más allá de este Proyecto, dentro del marco de trabajo del grupo de investigación UCASE.

3.5.1. WSDL2XSDTree

- Rediseñar la implementación con la pretensión de reducir aún más el número de ficheros generados para la construcción del árbol. Por ejemplo, reducir de dos a uno el número de ficheros en el caso de un documento WSDL que contiene un XML Schema embebido en la sección «types» con diferente `targetNamespace` al del elemento «definitions» del documento.

3.5.2. ServiceAnalyzer

- Expandir el sistema para que pueda considerar tipos *union*, *choice*, *all*, *mixed* y *wildcard*.

- Añadir una nueva orden que permita que la entrada sera un fichero BPEL y la herramienta de modo automático seleccione los ficheros WSDL a leer para generar el catálogo.

- Realizar un postprocesado de las declaraciones de tipos para simplificarlas en los casos que sea posible. Por ejemplo, si se ha definido un tipo lista de elementos de tipo t en la que el mínimo y el máximo número de elementos es uno, el postprocesado transformaría la declaración en la de un tipo simple t.

- Dotar a *ServiceAnalyzer* de interfaz gráfica para mejorar la interacción con el usuario.

CAPÍTULO 4

Desarrollo del proyecto

En esta sección vamos a describir las etapas del desarrollo de esta aplicación software: análisis, diseño, implementación y pruebas. En particular, las secciones de diseño y análisis describirán la última iteración de cada producto, al contener éstas la arquitectura y diseño definitivos, ya que han ido siendo refinados a lo largo de las iteraciones.

4.1. Metodología de desarrollo

Por las características del Proyecto en sí y del entorno de trabajo, dentro del grupo UCASE, se ha seguido una *metodología ágil* [11].

El desarrollo de software ágil hace referencia a un grupo de metodologías de desarrollo de software que se basan en principios y valores similares a los recogidos en el *Manifiesto Ágil*. Este trabajo pone en valor [1] :

- Los individuos y sus interacciones sobre los procesos y las herramientas.

[1] Aunque los elementos de la derecha tienen valor (procesos, herramientas, documentación y contratos), en los métodos ágiles se valoran más los de la izquierda (individuos, software, colaboración con el cliente y respuesta al cambio).

- El trabajo con el software sobre la documentación completa.
- La colaboración con el cliente sobre la negociación de contratos.
- La respuesta al cambio sobre el seguimiento de un plan.

En concreto, de entre los diferentes métodos ágiles, se ha seguido la metodología de XP (eXtreme Programming) [4, 35], que explicaremos en los siguientes apartados.

4.1.1. Origen

La metodología XP es un enfoque de la ingeniería del software formulado por Kent Beck. Es el más destacado de los procesos ágiles de desarrollo de software. Al igual que éstos, XP se diferencia de las metodologías tradicionales principalmente en que pone más énfasis en la adaptabilidad que en la previsibilidad.

Las raíces de la XP yacen en la comunidad de Smalltalk, y en particular la colaboración cercana de Kent Beck, Ward Cunningham y Ron Jeffries que, desde finales de los 80, fueron extendiendo sus ideas de un desarrollo de software adaptable y orientado a la gente.

El paso crucial de la práctica informal a una metodología ocurrió en la primavera de 1996. A Kent se le pidió revisar el progreso del proyecto de nómina C3 para Chrysler. El proyecto estaba siendo llevado en Smalltalk por una compañía contratista, y estaba en problemas. Debido a la baja calidad de la base del código, Kent recomendó empezar desde cero. El proyecto entonces reinició bajo su dirección, convirtiéndose en el campo de entrenamiento de la metodología XP.

En 1999, la popularidad de XP aumentó con la publicación el primer libro sobre la materia, “Extreme Programming Explained: Embrace Change”, escrito por Kent.

4.1.2. Características

XP es una metodología ágil de desarrollo de software, aplicable por lo general a proyectos de pequeño y medio tamaño, en un equipo de hasta 12 personas.

Los defensores de XP consideran que los cambios de requisitos sobre la marcha son un aspecto natural, inevitable e incluso deseable del desarrollo de proyectos. Creen que ser capaz de adaptarse a los cambios de requisitos en cualquier punto de la vida del proyecto es una aproximación mejor y más realista que intentar definir todos los requisitos al comienzo del proyecto e invertir esfuerzos después en controlar los cambios en los requisitos. No se trata de anticipar el cambio, sino de adaptarse a él.

En los métodos ágiles, se descarta todo documento o procedimiento innecesario para el proyecto, permitiendo al equipo avanzar rápidamente concentrándose en lo importante y sin perder tiempo en formalismos. En cada iteración, se entrega una versión funcional del software que el cliente puede probar para ampliar la información disponible en la siguiente iteración. Así, XP se dirige más a la obtención de software útil que al seguimiento de un proceso rígido y burocrático.

Una idea que se suele tener respecto a XP es que no se realiza documentación en absoluto. Sin embargo, un requisito muy común por parte del cliente, es la documentación externa al proyecto, por lo que debe ser atendido. Únicamente se descarta toda aquella documentación innecesaria, que sea un mero formalismo en lugar de una necesidad del cliente. Ron Jeffries aclara éste y

otros malentendidos en [34].

XP es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en el desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. Así, XP se dirige más a las personas que al proceso.

A diferencia de otras metodologías, donde el cliente es una entidad externa al proyecto con el que se sólo se comunica el analista, en XP es una parte integral del equipo. Se ocupa de formular los requisitos a nivel conceptual, darles su prioridad, y resolver dudas que pueda tener el resto del equipo.

XP nos recuerda así que el software existe para dar un valor añadido al cliente, y no por sí mismo. Así, las decisiones acerca de la funcionalidad deseada son del cliente, y las decisiones técnicas y el calendario lo establecen los desarrolladores.

4.1.3. Valores

Los valores básicos de XP son: simplicidad, comunicación, retroalimentación y coraje.

Simplicidad Se debe diseñar la solución más simple que pueda funcionar y ser implementada en un momento determinado del proyecto. Esto no quiere decir que se use la primera solución que se nos ocurra. La solución escogida debe:

- Hacer todo lo esperado.
- No contener ninguna duplicación.
- Pasar todas las pruebas.
- Tener el mínimo número de elementos.

Para mantener la simplicidad es necesaria la refactorización del código, ésta es la manera de mantener el código simple a medida que crece. También se aplica la simplicidad en la documentación, de esta manera el código debe comentarse en su justa medida, intentando que el código esté autodocumentado en la medida de lo posible, mediante nombres de funciones y variables descriptivos, por ejemplo.

Comunicación Consiste en la comunicación transparente y sin obstáculos entre todas las partes del equipo, incluido el cliente, aclarando confusiones y dudas en el momento. Los programadores se comunican constantemente gracias a la *programación por parejas*. La comunicación con el cliente es fluida ya que el cliente forma parte del equipo de desarrollo, decide qué características tienen prioridad y siempre debe estar disponible para solucionar dudas.

Retroalimentación Al realizarse ciclos muy cortos tras los cuales se muestran resultados, se minimiza el tener que rehacer partes que no cumplen con los requisitos y ayuda a los programadores a centrarse en lo que es más importante. Más que intentar capturar toda la información de una vez, un proyecto basado en XP debe de concentrarse en el día a día, aprendiendo paso a paso qué es lo que se espera exactamente del programa, dado que ni siquiera el cliente lo sabe con seguridad. Dicha información puede venir de muchas fuentes: del cliente, de la experiencia propia, de las pruebas automatizadas, etc.

Coraje Asumir retos, ser valientes antes los problemas y afrontarlos.

Para seguir esos valores, se proponen una serie de principios, que se concretan a través de una serie de prácticas recomendadas.

4.1.3.1. Principios

Entre los principios más importantes, podemos citar:

Flujo Se deben de realizar constantemente todas las actividades del desarrollo de software: análisis, diseño, implementación, integración y pruebas. Esto evita la acumulación de riesgos a la hora de integrar y validar, realizando entregas frecuentes con pequeños riesgos e incrementos de funcionalidad, más que una única entrega con alto riesgo y toda la funcionalidad.

Margen En todo plan, se debe de tener margen para, en el caso de no tener suficiente tiempo, poder retrasar la implementación de algunas historias a posteriores versiones. Como se ha visto en el capítulo 1, *ServiceAnalyzer* no contempla todas las posibilidades de XML Schema en la traducción de tipos, sin embargo, el producto es completamente funcional (salvando esas limitaciones) y mantiene un diseño que facilitará futuras ampliaciones.

Calidad En ningún momento se debe dejar de lado la calidad, es mejor hacer una única cosa bien que muchas, pero mal. Además es erróneo pensar que una reducción de la calidad conlleva una reducción del tiempo de desarrollo, es más, puede incluso aumentarlo.

Centrado en las personas Son las personas las que realizan el software, no la metodología de desarrollo.

4.1.3.2. Prácticas

Desde los comienzos de XP se le han asociado un gran número de buenas prácticas. En el caso de este Proyecto, por razones obvias, hay prácticas que

sencillamente no se han podido seguir, como “Programación en Parejas” [2] , o “Trabajar Juntos” [3]. Comentamos a continuación las que sí se han seguido:

Las historias de usuarios son la técnica utilizada para especificar los requisitos del software. Se trata de tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean requisitos funcionales o no funcionales. La tabla 4.1 representa un ejemplo del formato y contenido que pueden tener las historias de usuario.

Historia de usuario	
Número: 1	Nombre: Visualizar fichero
Modificación de historia nº:	Iteración asignada:
Prioridad en negocio: Media	Puntos estimados:
Riesgo en desarrollo: Baja	Puntos reales:
Descripción: 1. Voy al menú Fichero y le doy a “Abrir”. 2. En “Nombre de fichero”, introduzco “/tmp/mifichero.txt”. 3. Pulso en el botón “Abrir”. 4. Se abre el fichero “/tmp/mifichero.txt” y veo sus contenidos.	
Observaciones:	

Tabla 4.1: Ejemplo de tarjeta con historia de usuario

El tratamiento de las historias de usuario es muy dinámico y flexible. Cada historia de usuario es lo suficientemente comprensible y delimitada para que los programadores puedan implementarlas en unas semanas.

El equipo de desarrollo puede realizar estimaciones del esfuerzo que requeriría una historia y presentárselas al cliente, dándole más información para decidir qué historias quiere implementadas en cada iteración o si desea modificarla. En el caso del presente PFC, los “clientes” han

[2] Técnica de desarrollo de software en la que dos programadores trabajan juntos en el mismo ordenador. Uno (*driver*) teclea el código mientras que el otro (*observer*) revisa cada línea del código a medida que el primero lo va escribiendo. Se cambian los roles con frecuencia.

[3] La mayoría de los equipos ágiles se encuentran localizados en una única ubicación abierta para facilitar la comunicación.

sido los miembros de grupo UCASE, dedicados a las pruebas de composiciones de Servicios Web, especialmente los directores de proyecto.

Es una técnica completamente diferente de la de casos de uso, ya que una historia es algo mucho más concreto, generalmente pequeño y fácilmente estimable. Por su lado, un caso de uso se podría ver como una clase de interacciones concretas que el sistema debe plasmar. Ambos describen el qué, pero lo hacen bajo distintos enfoques. XP contempla la utilización de los diagramas de casos de usos como una posible herramienta de apoyo, que sirva de generalización de una o varias historias de usuario particulares.

Integración continua La prueba e integración de los cambios se realiza de forma continua y a pequeños pasos. Deben tenerse siempre un ejecutable del proyecto que funcione y en cuanto se tenga una nueva pequeña funcionalidad, debe recompilarse y probarse. Es un error mantener una versión congelada durante mucho tiempo mientras se hacen mejoras y luego integrarlas todas de golpe. Cuando falle algo, no se sabe qué es lo que falla de todo lo que hemos metido. Cuanto menores sean los cambios, menor será el coste en tiempo y esfuerzo que se dedique a corregir los fallos introducidos con los cambios.

Ciclo semanal La planificación se ha realizado en ciclos cortos, de duración aproximada de una semana. Durante los mismo se ha mantenido una comunicación frecuente con los directores del proyecto, vía correo electrónico y en persona.

Compilación en 10 minutos Consiste en mantener siempre el tiempo de compilación y ejecución de todas las pruebas automáticas pertinentes por debajo de 10 minutos, para poder realizar integración continua del código

de todo el equipo y evitar tanto problemas de integración de última hora como esperas innecesarias.

Dada la envergadura de este Proyecto, esta práctica se ha conseguido de manera natural.

Desarrollo orientado a pruebas (TDD) Técnica de desarrollo de software [4] que usa iteraciones de desarrollo cortas basadas en casos de prueba escritos previamente que definen las mejoras deseadas o nuevas funcionalidades [3]. Surgió de la mano de XP, pero actualmente acapara un interés general mayor en sí mismo.

Además de garantizar mayor corrección, el uso de este estilo obliga al desarrollador a estructurar su código para que sea fácil de probar, haciéndolo modular y cohesivo. Refactorizaciones posteriores en el diseño pueden garantizar inmediatamente la misma funcionalidad que la versión anterior.

Dada la importancia de las pruebas en este PFC, está técnica ha sido de gran ayuda en el desarrollo de los principales componentes de *WSDL2XSDTree* y *ServiceAnalyzer*. Para la automatización de las pruebas se ha usado `JUnit`. La secuencia seguida cada vez que queremos dotar al sistema de una nueva característica es la siguiente:

1. Diseñar y añadir una prueba: esta prueba deberá fallar inevitablemente porque se escribe antes de que se haya implementado la característica. Para escribir una prueba, el desarrollador debe entender claramente la especificación y los requisitos de la característica. El desarrollador puede llevar a cabo esto a través de casos de uso e

[4] Hay que tener en cuenta que TDD (Test-Driven Development) es un método de diseño de software, no sólo un método de pruebas.

historias de usuario que cubran los requisitos y las condiciones de excepción.

2. Ejecutar las pruebas y comprobar que la última que se ha añadido falla: esto valida que las pruebas están funcionando correctamente y que las pruebas añadidas no pasan inesperadamente, sin la necesidad de código nuevo.

3. Implementar la característica: en este paso se añade o modifica el código para que haga que se puedan pasar las pruebas añadidas. El código escrito en esta etapa no será perfecto y puede, por ejemplo pasar la prueba de una forma poco elegante. Esto es aceptable porque en pasos sucesivos se mejorará y perfeccionará. Es importante resaltar que el código escrito sólo se diseña para pasar la prueba; no se debería predecir más funcionalidad.

4. Ejecutar las pruebas automatizadas: si todas las pruebas tienen éxito, el programador puede estar seguro de que el código cumple todos los requisitos probados. En caso contrario, retrocederá al paso anterior para hacer las modificaciones pertinentes, y así hasta que el código pase todas las pruebas.

5. Mejorar el diseño: se refactoriza el código para mejorar la calidad del diseño. Una vez modificado el código, se vuelven a ejecutar las pruebas para comprobar que la refactorización no ha dañado ninguna funcionalidad existente.

Diseño incremental Las metodologías tradicionales, siguiendo el modelo de ciclo de vida de cascada, intentaban realizar el análisis y diseño de antemano, siguiendo el modelo de la ingeniería tradicional. XP afirma lo contrario: el diseño del programa no es algo que se haga una sola vez

y quede fijo por el resto de la vida del programa. Se debe refinar continuamente en función de las necesidades del momento. Para evitar el aumento del coste de las modificaciones, se dispone de otras prácticas además de ésta, como el uso de pruebas automáticas, la compartición de código, la programación en parejas y la comunicación fluida con el resto del equipo. Así evitamos tanto el diseño excesivo, que supone una pérdida de tiempo, como la falta de diseño, que dificulta las modificaciones.

4.2. Herramienta de modelado usada: BOUML

BOUML es una herramienta CASE [5] de UML (Unified Modeling Language) gratuita. Genera código Java/C++/Python/Perl a partir de diagramas UML, realiza ingeniería inversa de Java/C++, y permite dibujar diversos diagramas UML 2.0, como los diagramas de clases, despliegue, componentes o casos de uso, entre otros.

Está disponible para Windows, GNU/Linux y Mac OS X en la web `http://bouml.free.fr`. Se trata de un programa escrito en C++ usando la versión 3.3.8 de la biblioteca Qt de la compañía Trolltech, conocida por ser la biblioteca sobre la cual se halla implementado el gestor de escritorio KDE.

4.3. Especificación de los requisitos del sistema

Para el éxito de un desarrollo de software es esencial comprender totalmente los requisitos del software [51, 48]. Poca importancia tiene lo bien

[5] Las herramientas CASE son aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software reduciendo el coste de las mismas en términos de tiempo y de dinero. Estas herramientas nos pueden ayudar en todos las fases del ciclo de vida de desarrollo del software en tareas como el proceso de realizar un diseño del proyecto, cálculo de costes, implementación de parte del código automáticamente con el diseño dado, compilación automática, documentación o detección de errores entre otras.

diseñado o codificado que esté un programa si no se ha analizado correctamente.

Un requisito puede definirse como una condición que debe cumplir o poseer un sistema o alguno de sus componentes para satisfacer un contrato, una norma o una especificación o, dicho de otra manera más simple, una propiedad que debe tener para poder solucionar un problema del mundo real.

4.3.1. Requisitos de interfaces externas

En este apartado debe incluirse una descripción detallada de los requisitos de conexión a otros sistemas (hardware o software) con los que el sistema debe interactuar.

Como se ha mencionado ya, el usuario podrá utilizarla integrada en la herramienta `GAmera` o puede que la utilice de forma independiente.

También se encuadran en este apartado los requisitos de interfaz de usuario. El sistema debe tener:

- Una interfaz sencilla: el manejo y entendimiento de la aplicación debe ser sencillo e intuitivo. El perfil de usuario más habitual será un estudiante o profesor que colabore en un proyecto del grupo de investigación UCASE. El proyecto debe ir orientado a los WS de tal modo que se deba prestar atención a los mensajes que se intercambian entre los servicios y los *mockups*, bien para la prueba de la composición, bien para otros menesteres.

- Un manual que sirva de apoyo: es importante para complementar el requisito anterior, mostrar ayuda clara y concisa que facilite la interacción del usuario con el producto.

4.3.2. Requisitos funcionales

- Analizar las interfaces de una composición de Servicios Web descritas en WSDL 1.1.
- Generar un catálogo en formato XML que contemple y organice todos los posibles mensajes a intercambiar en una determinada composición de servicios.
- Construir plantillas paramétricas que permitan su uso posterior en varios casos de prueba dándole distintos valores a las variables que hacen posible esta parametrización.
- Traducir los elementos/tipos XML Schema implicados en los mensajes a los correspondientes de un nuevo sistema de tipos simplificado.
- Permitir la recepción de varios ficheros WSDL e incluir los mensajes de todos los servicios bajo un mismo catálogo.
- Generar un árbol de definiciones XML Schema a partir de un fichero WSDL de entrada. El árbol debe incluir todos los esquemas definidos o importados en el propio documento WSDL, así como los de los árboles correspondientes a los ficheros WSDL importados por el original, y así recursivamente.
- Detectar el incumplimiento de alguna de las principales especificaciones del WS-I Basic Profile 1.1 en la interfaz del servicio/s proporcionada como entrada.
- Asegurar que las plantillas generadas cumplen con las restricciones impuestas por WSDL, SOAP, XML Schema y el WS-I Basic Profile 1.1.

4.3.3. Requisitos de rendimiento

En cuanto al rendimiento, se requiere que la aplicación tenga un alto rendimiento, sobre todo, para que al integrarse en `GAmera` no incremente la sobrecarga del sistema.

4.3.4. Atributos del sistema software

El sistema debe tener:

- Independencia de plataforma: este requisito, también conocido como transportabilidad, no es imprescindible. Se corresponde con el deseo de permitir un libre uso de la herramienta que limite en lo mínimo posible la elección de sistema operativo.

- Mantenibilidad: es fundamental para que posteriores ampliaciones y correcciones se hagan de forma rápida y sencilla, dada la complejidad de la herramienta cuyo uso se pretende facilitar.

- Corrección: todas las plantillas y las respectivas declaraciones de variables deben generarse correctamente. En caso contrario, los casos de prueba construidos a partir de éstas no serían válidos.

- Facilidad de uso: uno de los objetivos de *ServiceAnalyzer* es liberar al usuario de tener que generar manualmente las plantillas para cada uno de los mensajes y las declaraciones de tipos, un proceso incómodo y propenso a errores difíciles de depurar debido a las restricciones que imponen WSDL, SOAP, XML Schema y el WS-I Basic Profile 1.1. Una vez cumplido este objetivo no podemos complicar inútilmente la vida del usuario con una interfaz complicada y poco intuitiva.

- Facilidad de instalación: controlando el número de dependencias externas y, cuando esto no sea posible, reduciendo su impacto en la complejidad de la instalación.

- Licencia libre: se desea que la aplicación tenga el mejor uso público posible y ésta es la mejor manera de conseguirlo.

4.4. Análisis del sistema

4.4.1. Historias de usuario

Como ya se adelantó en §4.1.3.2, los proyectos basados en XP se estructuran en torno a las historias de los usuarios, que describen un comportamiento o propiedad deseada del sistema en sus propias palabras.

Mientras se redactan, el equipo de desarrolladores, tras realizar un análisis superficial del trabajo implicado con ayuda del cliente, asigna a cada historia un tiempo estimado de finalización en días ideales y un riesgo. El cliente puede, en función de esos factores y de sus intereses, marcar la prioridad de una historia.

Una historia de usuario puede ser de muy alto nivel o de muy bajo nivel, pudiendo unirse varias historias de bajo nivel en una sola, o dividirse una de alto nivel en varias más sencillas.

Pasaremos a listar las historias de usuario que se han ido acumulando a lo largo de la comunicación con los clientes durante el desarrollo del PFC, es decir, en los seminarios con el grupo UCASE y las tutorías con los tutores.

En el caso del presente PFC, la interacción con el usuario es mínima, ya que, pese al trabajo de implementación que hay por detrás, el programa tiene una finalidad muy concreta de cara al usuario. Por ello, tenemos pocas historias de usuario y muy cortas.

El resto de requisitos especificados por los tutores del Proyecto (los clientes) se plasman en las reglas de reescritura que se expondrán más adelante, pero no son propiamente historias de usuario.

4.4.1.1. Generación de un árbol XSD

- ID: 1
- Nombre: “Generación de un árbol XSD a partir de un fichero WSDL”
- Descripción:
  1. En una terminal, voy al directorio donde tengo mi WSDL.
  2. Compruebo si el fichero en cuestión importa otros ficheros WSDL o XML Schema. Si es así verifico que se encuentran en la dirección relativa a la que se hace referencia al importar.
  3. Introduzco:

     ```
     WSDL2XSDTree f.wsdl
     ```

  4. Se crean en el directorio los documentos XSD correspondientes. Mínimo habrá un fichero XSD raíz con el nombre “f.wsdl.xsd”.
- Complejidad: Media
- Esta historia implica:
  - Realizar una transformación del documento WSDL de entrada que:
    - Ignore todas las partes del WSDL excepto los «import» y la sección «types».
    - Transforme los «WSDL:import» en «XSD:import».
    - Mantenga los «XSD:import».

- Concatene en el fichero XSD raíz resultante todos los XML Schema embebidos en la sección «types».
- Cree nuevos ficheros (nodos hijos) con aquellos XML Schema que difieran en `namespace` con el del XML Schema raíz (que tendrá el `targetnamespace` de la sección «definitions»).

- La implementación de una clase que llame recursivamente a la anterior, aplicando sucesivamente la transformación a los documentos WSDL importados por el fichero inicial.

Esta historia de usuario se ha dividido en varias en función del paso 2. Dichas historias pueden darse combinadas [6]. Son las siguientes:

- ID: 1.1
- Nombre: **"Generación de un árbol XSD a partir de un fichero WSDL independiente"**
- Descripción:
  1. En una terminal, voy al directorio donde tengo mi fichero "f.wsdl", que no importa otros ficheros.
  2. Introduzco:

     ```
     WSDL2XSDTree f.wsdl
     ```

  3. Se crean los documentos XSD que conforman el árbol. El fichero raíz será el creado con nombre "f.wsdl.xsd" en el mismo directorio de "f.wsdl".

- ID: 1.2

[6] Por ejemplo, la combinación de 1.2 y 1.4 generaría un árbol XSD con raíz "f.wsdl.xsd" y ramas "g.wsdl.xsd" y "f.wsdl_1.xsd"

- Nombre: **"Generación de un árbol XSD a partir de un fichero WSDL que importa otro WSDL"**
- Descripción:
  1. En una terminal, voy al directorio donde tengo mi fichero "f.wsdl" que importa a "g.wsdl".
  2. Introduzco:

     ```
     WSDL2XSDTree f.wsdl
     ```

  3. Se crean los documentos XSD que conforman el árbol. Mínimo habrá dos ficheros XSD: el fichero raíz con nombre "f.wsdl.xsd" y el fichero "g.wsdl.xsd" que es importado por el primero. El fichero raíz se crea en el directorio actual, mientras que "g.wsdl.xsd" se crea en el mismo directorio de "g.wsdl".
- ID: 1.3
- Nombre: **"Generación de un árbol XSD a partir de un fichero WSDL que importa un fichero XSD"**
- Descripción:
  1. En una terminal, voy al directorio donde tengo mi fichero "f.wsdl" que importa a "g.xsd".
  2. Introduzco:

     ```
     WSDL2XSDTree f.wsdl
     ```

  3. Se crean los documentos XSD que conforman el árbol. El árbol de salida lo compondrán al menos dos ficheros XSD: el fichero raíz con nombre "f.wsdl.xsd" y el fichero "g.xsd", que ya existía pero que ahora es importado por el fichero raíz.

- ID: 1.4
- Nombre: **“Generación de un árbol XSD a partir de un fichero WSDL que concatena esquemas XML con diferentes `targetNamespace`”**
- Descripción:
  1. En una terminal, voy al directorio donde tengo mi fichero “f.wsdl” que contiene en «types» un XML Schema con diferente `targetNamespace` que el del elemento «definitions».
  2. Se crean los ficheros XSD que conforman el árbol: el fichero raíz con nombre “f.wsdl.xsd” y el fichero “f.wsdl_1.xsd”, que es importado por el primero.

4.4.1.2. Generación del catálogo de mensajes

- ID: 2
- Nombre: “Generación del catálogo de mensajes correspondiente a un fichero WSDL”
- Descripción: Esta historia de usuario es muy corta para todo el trabajo que implica y consta de los siguientes pasos:
  1. En una terminal, voy al directorio donde tengo mi WSDL. Si el fichero en cuestión importa otros ficheros WSDL o XML Schema, deben estar en la dirección relativa a la que se hace referencia.
  2. Introduzco:

     ```
     WS f.wsdl
     ```

  3. Obtengo por salida estándar el catálogo en formato XML.
- Complejidad: Alta

- Esta historia implica:
  - Definición de la estructura de la salida del programa.
  - Construcción de una representación en memoria del catálogo de mensajes.
  - Lectura de los ficheros WSDL.
  - Lectura y análisis del contenido XML Schema.
  - Construcción de una representación en memoria del sistema de tipos definido por todos los ficheros WSDL y XML Schema necesarios.
  - Generación de plantillas.
  - Generación de declaraciones.
  - Conversión del catálogo generado al formato de salida.

De esta historia de usuario se dedujo otra, que puede considerarse como una ampliación de la anterior.

- ID: 3
- Nombre: **"Generación del catálogo de mensajes correspondiente a una composición"**
- Descripción: Esta historia de usuario es muy corta para todo el trabajo que implica y consta de los siguientes pasos:
  1. En una terminal, voy al directorio donde tengo los ficheros WSDL de mi composición.
  2. Introduzco:

     ```
     ServiceAnalyzer f.wsdl otro/g.wsdl
     ```

3. Obtengo por salida estándar el catálogo en formato XML con los mensajes de todos los servicios descritos en “f.wsdl” y “g.wsdl”.

4.4.2. Casos de uso

La notación gráfica que utiliza UML para representar los casos de uso es lo que conocemos como diagramas de casos de uso. Los diagramas correspondientes a los dos sistemas del presente PFC se representan mediante las figuras 4.1 y 4.2. No obstante, los diagramas de casos de usos son una mera representación, lo realmente importante en el modelado de casos de uso es la especificación de éstos.

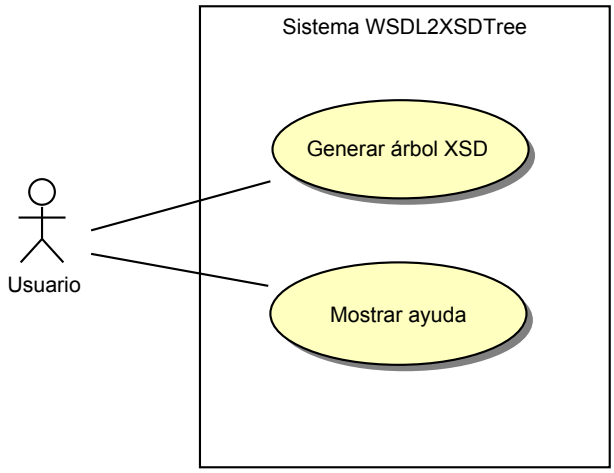

Figura 4.1: Diagrama de casos de uso de WSDL2XSDTree

4.4.2.1. Generar árbol XSD

- Actor principal: Usuario que desea generar un árbol de definiciones XML Schema a partir de un fichero WSDL.
- Precondiciones: Existe el documento WSDL proporcionado en la ruta especificada.

- Postcondiciones: Se crea un fichero con extensión XSD en el mismo directorio del documento WSDL de entrada que será la raíz del árbol XSD generado. Dependiendo de la estructura de los nodos del árbol pueden crearse otros ficheros XSD.
- Escenario principal:
  1. El usuario indica su intención de generar el árbol XSD mediante la orden:

     `WSDL2XSDTree ruta`

     donde `ruta` es la ruta al fichero WSDL.
  2. El sistema analiza el contenido del fichero WSDL para organizar el sistema de tipos definido en forma de árbol, creando los ficheros necesarios.
- Variaciones:

  1a. El nombre de la orden es incorrecto.
  1. El sistema muestra un mensaje informando de que no se conoce la orden introducida y cancela el caso de uso.

  1b. El número de argumentos introducidos en la línea de órdenes es incorrecto.
  1. El sistema muestra un mensaje informando de que el número de argumentos es incorrecto e imprime por pantalla el modo de uso de la herramienta. Se cancela el caso de uso.

4.4.2.2. Mostrar ayuda de WSDL2XSDTree

- Actor principal: Usuario que desea obtener la ayuda de la herramienta *WSDL2XSDTree*.

- Precondiciones: Ninguna.
- Postcondiciones: Se muestra en la terminal la información correspondiente al nombre y la versión del programa así como las opciones de uso.
- Escenario principal:
  1. El usuario indica su intención de obtener la ayuda del programa.
  2. El sistema muestra la ayuda por pantalla.

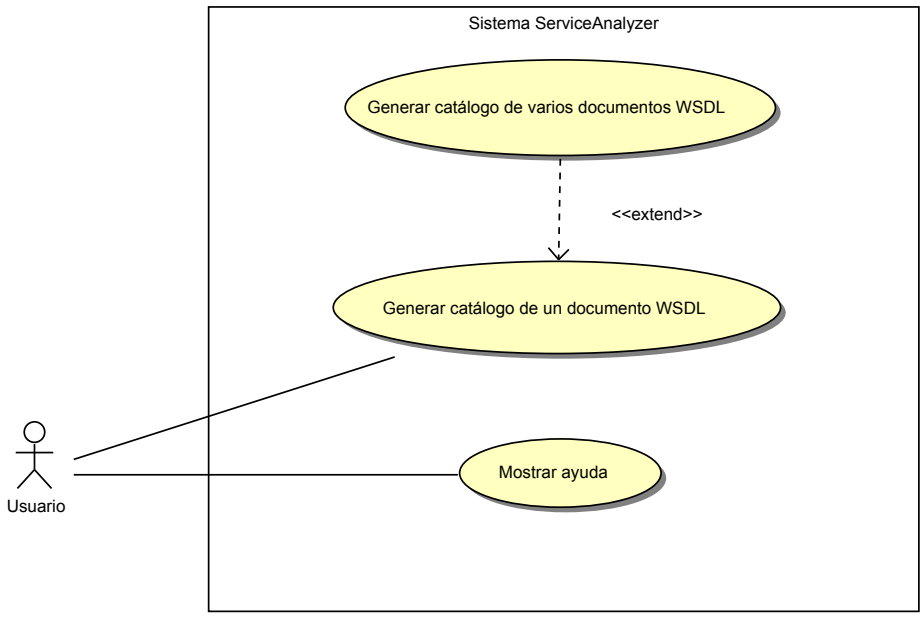

Figura 4.2: Diagrama de casos de uso de ServiceAnalyzer

4.4.2.3. Generar catálogo de un documento WSDL

- Actor principal: Usuario que desea generar el catálogo de mensajes correspondiente a la interfaz definida en un fichero WSDL.

- Precondiciones: Existe el documento WSDL proporcionado en la ruta especificada.
- Postcondiciones: Se muestra por pantalla el catálogo con todos los posibles mensajes que pueden darse.
- Escenario principal:

  1. El usuario indica su intención de generar el catálogo mediante la orden:

     `ServiceAnalyzer ruta`

     donde `ruta` es la ruta al fichero WSDL.
  2. El sistema analiza el contenido del fichero WSDL y genera un catálogo en formato XML con las plantillas y las declaraciones de variables correspondientes a cada uno de los posibles mensajes. Finalmente, muestra el catálogo por pantalla.

- Variaciones:

  1a. El nombre de la orden es incorrecto.

  1. El sistema muestra un mensaje informando de que no se conoce la orden introducida y cancela el caso de uso.

  1b. El número de argumentos introducidos en la línea de órdenes es incorrecto.

  1. El sistema muestra un mensaje informando de que el número de argumentos es incorrecto e imprime por pantalla el modo de uso de la herramienta. Se cancela el caso de uso.

  2a. El WSDL es inválido o no respeta el WS-I Basic Profile 1.1, o algún XML Schema no compila.

1. El sistema muestra el mensaje de error correspondiente y cancela el caso de uso.

4.4.2.4. Generar catálogo de varios documentos WSDL

- Actor principal: Usuario que desea generar el catálogo de mensajes correspondiente a la interfaz definida en una serie de ficheros WSDL.
- Precondiciones: Existen los documentos WSDL proporcionados en la ruta especificada.
- Postcondiciones: Se muestra por pantalla un único catálogo con todos los posibles mensajes que pueden darse.
- Escenario principal:
  1. El usuario indica su intención de generar el catálogo mediante la orden:

     `ServiceAnalyzer ruta`

     donde `ruta` es la ruta al fichero WSDL.
  2. El sistema repite el proceso realizado en el caso de uso anterior hasta haber analizado todos los documentos de entrada y finalmente, genera el catálogo con el resultado.

4.4.2.5. Mostrar ayuda de ServiceAnalyzer

- Actor principal: Usuario que desea obtener la ayuda de la herramienta *ServiceAnalyzer*.
- Precondiciones: Ninguna.

- Postcondiciones: Se muestra en la terminal la información correspondiente al nombre y la versión del programa y las opciones de uso.

- Escenario principal:

  1. El usuario indica su intención de obtener la ayuda del programa.
  2. El sistema muestra la ayuda por pantalla.

4.4.3. Modelo de datos del dominio de WSDL2XSDTree

4.4.3.1. Notación

En la figura 4.3 de la página 97 se presenta el diagrama de clases conceptuales para *WSDL2XSDTree*. Se ha utilizado la notación UML 2.0.

4.4.3.2. Descripción general

La ruta de un DOCUMENTOWSDL es lo que se debe proporcionar como entrada a la herramienta. Como podemos observar en la figura 4.3, un DOCUMENTOWSDL será transformado en un ÁRBOLXSD. Dicho ÁRBOLXSD estará formado por al menos un DOCUMENTOXSD, que representará la *raíz* del árbol. El resto de nodos del ÁRBOLXSD, si los hay, serán agregados a partir del DOCUMENTOXSD *raíz*.

En la transformación hay que tener presente que cada DOCUMENTOWSDL puede contener en el elemento TYPES varios XMLSCHEMA embebidos, concatenados uno detrás de otro. En este elemento también podemos encontrar elementos «import» que hagan referencia a otros XMLSCHEMA guardados en objetos de la clase DOCUMENTOXSD externos, que a su vez podrían importar otros. El elemento «types» de un DOCUMENTOWSDL también puede estar vacío.

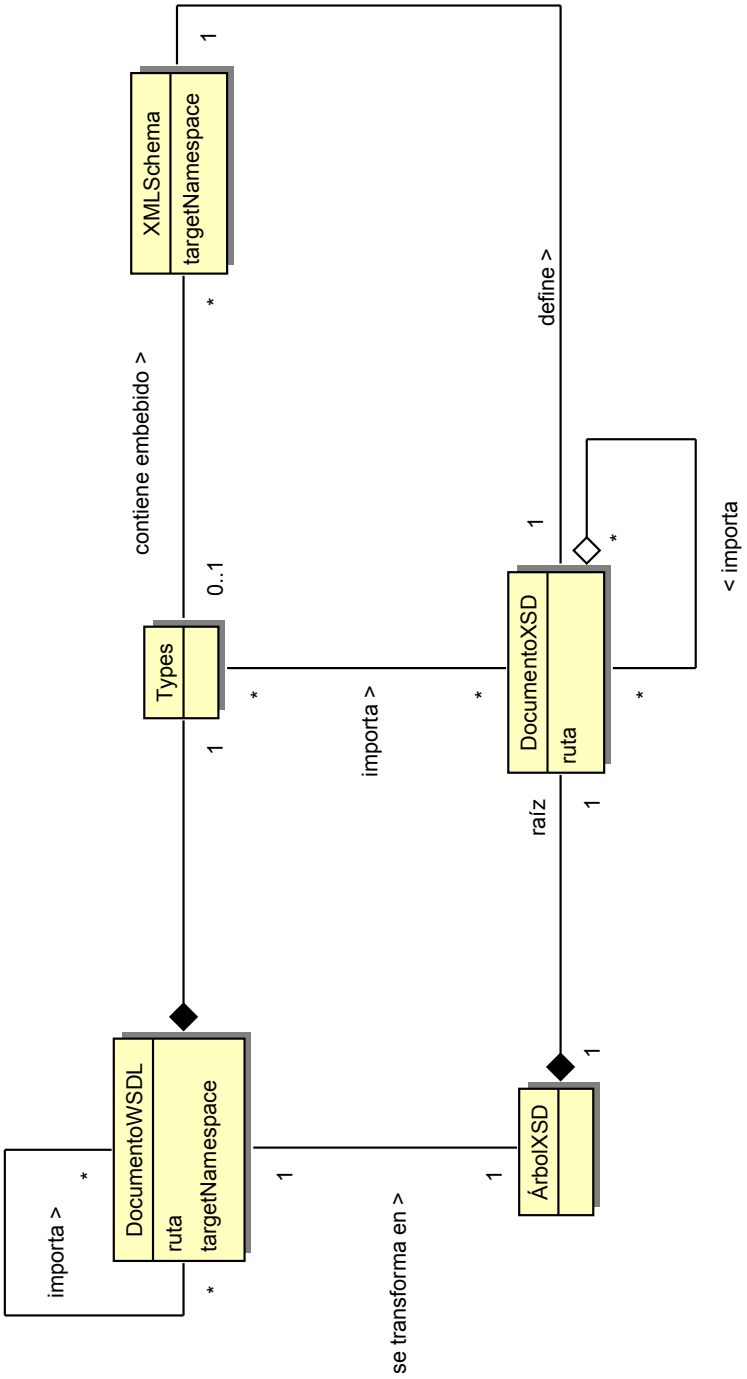

Figura 4.3: Diagrama de clases conceptuales de WSDL2XSDTree

Por otro lado, un DOCUMENTOWSDL también puede que importe otros ficheros WSDL, por lo que la transformación debe ser recursiva.

4.4.4. Modelo de datos del dominio de ServiceAnalyzer

4.4.4.1. Notación

En la figura 4.4 se presenta el diagrama de clases conceptuales para este sistema, también en notación UML 2.0. La clase DOCUMENTOWSDL aparece también en el diagrama anterior. Para evitar confusiones, la clase duplicada se muestra en color azul y se han ocultado sus atributos.

4.4.4.2. Descripción general

La interfaz de un WS se describe a través de una serie de instancias de DOCUMENTOWSDL. Estos documentos son los que va a recibir *ServiceAnalyzer* como entrada para generar el CATÁLOGO de MENSAJES.

El CATÁLOGO va a estar compuesto por los servicios, representados por la clase SERVICIOS en la figura 4.4, que se definen en los WSDL proporcionados como entrada. Cada SERVICIO se identificará por su nombre y su URI. Por cada elemento SERVICIO vamos a tener una serie de elementos PUERTO, cada uno con cero o más elementos OPERACIÓN.

Las operaciones definen la forma de comunicarse para consumir los servicios. En esta comunicación se intercambian una serie de MENSAJES. Cada OPERACIÓN puede tener un mensaje de solicitud (clase ENTRADA) o un mensaje de respuesta (clase SALIDA) o ambos. A su vez, puede contener elementos FALLO, que representan los mensajes de error.

Por cada MENSAJE que se incluya en el CATÁLOGO vamos a tener una PLANTILLA que permite generar distintas instancias del mismo. Para ello, la PLANTILLA se describe en función de una serie de VARIABLES. Cada VARIABLE tiene

asociado un nombre que la identifica de forma única, así como un tipo. El tipo define los valores que puede tomar dicha variable. El tipo puede ser uno de los predefinidos por el sistema empleado o bien, un TYPEDEF.

Un TYPEDEF es una variable "especial" que no se utiliza en la PLANTILLA pero que sirve para crear nuevos tipos que serán empleados por las VARIABLES. Un TYPEDEF puede definirse a partir de un tipo primitivo o de otro TYPEDEF. Al tipo sobre el cual se define se le pueden añadir una serie de restricciones que se describirán en §4.5.2.

A cada elemento PLANTILLA se le asocia un bloque con las DECLARACIONES de VARIABLES y de los TYPEDEFS que se necesitan para describir el tipo de dichas VARIABLES. Dichas declaraciones también se incluyen en el CATÁLOGO.

4.4.4.3. Restricciones textuales

- El atributo `type` de una instancia de la clase TYPEDEF tendrá asignado el valor del nombre de uno de los tipos predefinidos. Actualmente se dispone de "int", "float", "string", "duration", "date", "datetime", "time", "list" y "tuple".
- Si el atributo `type` de un TYPEDEF tiene asignado el valor "list", entonces el atributo `element` debe estar relleno y contener el nombre de otro TYPEDEF o uno de los tipos predefinidos.
- Si el atributo `type` de un TYPEDEF tiene asignado el valor "tuple", entonces el atributo `element` debe estar relleno y contener una lista de nombres de otros TYPEDEF y/o de nombres de algunos de los tipos predefinidos.
- Los atributos `min`, `max`, `values`, `pattern`, `totalDigits` y `fractionDigits` de TYPEDEF son opcionales, pueden no estar rellenos.

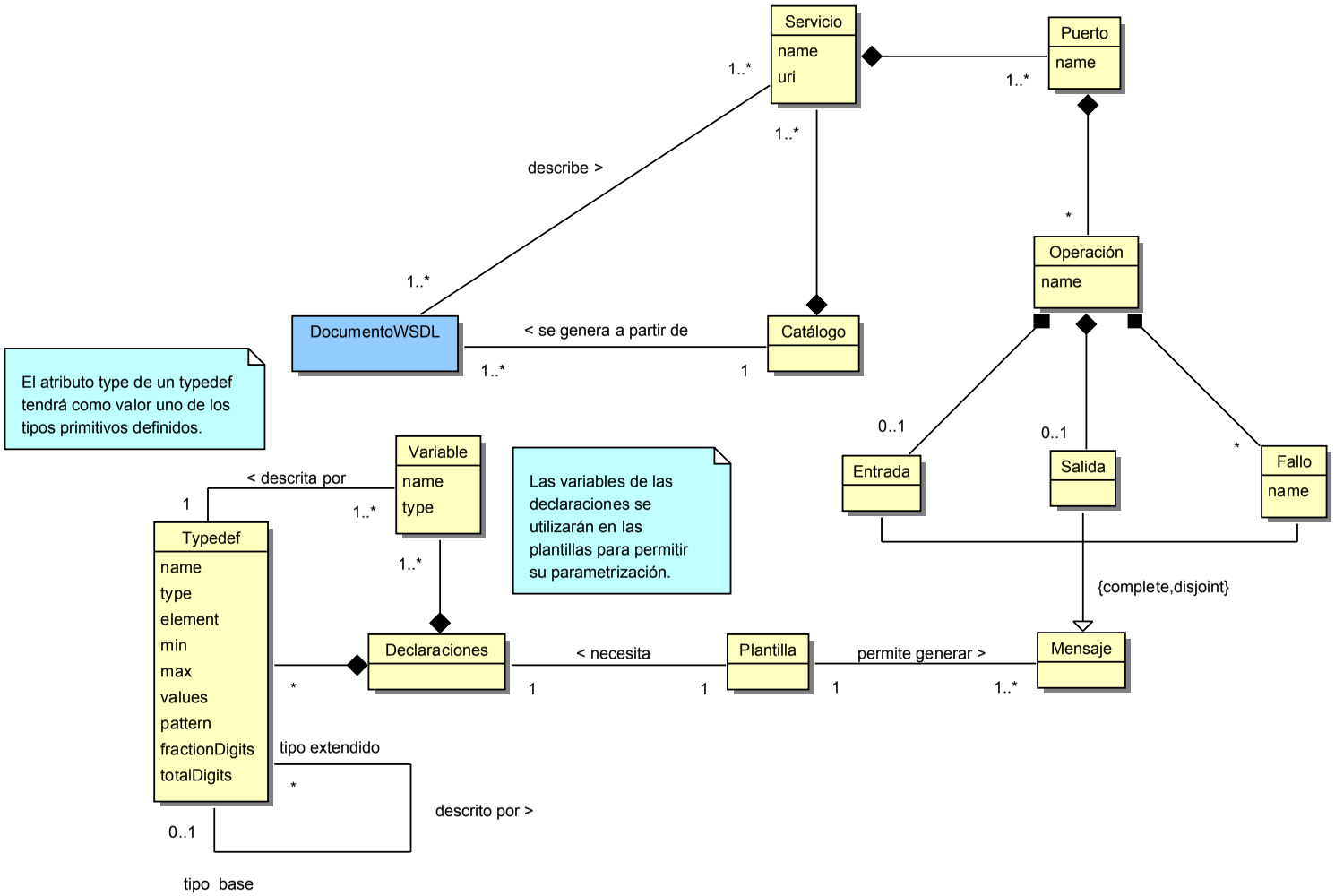

Figura 4.4: Diagrama de clases conceptuales de ServiceAnalyzer

4.5. Diseño del sistema

Para poder decidir la arquitectura del sistema, primero habría que definir cuáles son las condiciones que afectarán a su estructura.

Por supuesto, la estructura va a venir determinada primordialmente por la naturaleza del problema. En los siguientes apartados se van a comentar los aspectos del diseño que han marcado el desarrollo del Proyecto.

Por otro lado, otro condicionante que determina la arquitectura del sistema es la búsqueda de la calidad del software: nuestro objetivo es crear un software flexible, fácilmente mantenible, que permita la reutilización de sus componentes y que sea fiable.

4.5.1. Estructura del catálogo

La primera idea de la aplicación contemplaba que *ServiceAnalyzer* generase dos ficheros:

- Uno en formato XML con las plantillas clasificadas según el servicio y puerto al que perteneciera la operación y si se trataba de un mensaje de entrada, salida o error.

Listado 4.1: Primer boceto del formato de salida de las plantillas

```
1  <services>
2    <service name="..." uri="...">
3          <port name="...">
4              <operation name="...">
5               <input>
6                 <met:MetaSearchProcessRequest xmlns:met="...">
7                  <met:query>$query</met:query>
8                  <met:language>$language</met:language>
9                  <met:country>$country</met:country>
10                 <met:maxResults>$maxResults</met:maxResults>
```

```
11                  </met:MetaSearchProcessRequest>
12                </input>
13                <output>
14                </output>
15                <fault name="...">
16                </fault>
17              </operation>
18              <operation ...>
19              ...
20            </port>
21  </service>
22 </services>
```

- Otro fichero con formato plano en el que se hicieran las declaraciones de variables con:
  - Tantas filas como variables.
  - Una primera columna para los nombres de variables.
  - Un segunda columna con el tipo de las variables.
  - Una tercera columna en la que describir las restricciones asociadas.

Listado 4.2: Primer boceto del formato de salida de las declaraciones

```
1  natural integer min = 0
2  query string
3  language string
4  country string
5  maxResults natural
6
7  req_amount integer min = 0, max = 255
8  ap_reply string values = ['true':'false']
9  as_reply string values = ['low':'high']
10 total float none
```

```
11
12 lista_cadenas list(string) list_min = 1, list_max = 5, values = ['low':'high']
```

Pero un análisis más a fondo reveló los problemas de usar dos ficheros:

- ¿Cómo relacionamos las declaraciones con las distintas partes del catálogo?
- ¿Cómo separamos esos conjuntos de declaraciones entre sí?

La solución encontrada fue añadir esa información de tipos al propio catálogo, y luego generar el fichero de tipos para `GAmera` con el formato que queramos a partir de él. Esta solución además de solventar los problemas anteriores ofrece una gran ventaja: permite cambiar fácilmente el formato de entrada de `GAmera` sin tener que modificar demasiado *ServiceAnalyzer*.

Al incluirse las declaraciones en el catálogo, no basta con "copiar y pegar" los datos del fichero de variables en el del catálogo de plantillas, sino que ha de transformase el contenido al formato XML e incluirse en una nueva etiqueta. En el listado 4.3 podemos ver cómo quedaría el bloque «input» del ejemplo anterior.

Listado 4.3: Boceto del formato final con una única salida

```
1  <input>
2      <decls>
3          <typedef name="natural" type="integer" min="0"/>
4          <variable name="query" type="string"/>
5          <variable name="language" type="string"/>
6          <variable name="country" type="string"/>
7          <variable name="maxResults" type="natural"/>
8      </decls>
9      <template>
10      <met:MetaSearchProcessRequest xmlns:met="...">
```

```
11            <met:query>$query</met:query>
12            <met:language>$language</met:language>
13            <met:country>$country</met:country>
14            <met:maxResults>$maxResults</met:maxResults>
15           </met:MetaSearchProcessRequest>
16          <template>
17  </input>
```

4.5.2. Sistema de tipos definido

Como ya sabemos, las plantillas se van a parametrizar en base a una serie de variables. Las declaraciones de tipo de estas variables serán necesarias para un futuro componente que se encargará de generar valores válidos para dichas variables. De este modo, podremos obtener un conjunto de casos de prueba para el mismo mensaje y automatizar la prueba de dicho conjunto.

La idea es usar un sistema de tipos simplificado cuyos valores sean más fáciles de generar que los del complejo sistema de tipos de WSDL. WSDL permite el uso de diversos sistemas de tipos, siendo XML Schema el de un uso más generalizado. Por ello, nos hemos centrado en éste en particular.

A continuación, procedemos a describir el sistema de tipos utilizado. En primer lugar, se han definido una serie de tipos primitivos para nuestro sistema. Éstos son:

- ***string***: Cadena de caracteres válidos Unicode e ISO/IEC 10646.
- ***int***: Enteros que pueden almacenarse en 32 bits.
- ***float***. Representa números de coma flotante de 32 bits, según el estándar IEEE 754.
- ***date***. Define un día concreto del calendario Gregoriano con el formato `YYYY-MM-DD`, como por ejemplo, “2003-10-21”.

- ***time***. Define una hora concreta con el formato `hh:mm:ss`, como por ejemplo, "10:21:23". El número de segundos puede incluir dígitos decimales de precisión arbitraria si se desea. Las horas se datan según el sistema de 24 horas.

- ***dateTime***. Define un instante de tiempo concreto, usando el calendario Gregoriano. El formato es `YYYY-MM-DDThh:mm:ss`, por ejemplo, "2003-10-21T20:30:13". La `T` separa la fecha de la hora. Se puede añadir también una `Z` opcional, y $\pm$ `hh:mm` al final para indicar una zona horaria diferente. Usamos `Z` si la hora se ajusta GMT (Greenwich Mean Time) o o a UTC (Coordinate Universal Time), o utilizaremos las horas y minutos adicionales para indicar diferencias respecto al GMT.

- ***duration***. Expresa una duración en un espacio de 6 dimensiones. El formato es `PnYnMnDTnHnMnS`, donde `nY` representa el número de años, `nM` el número de meses, `nD` el número de días, `nH` el número de horas, `nM` el número de minutos y `nS` el de segundos. `P` es un indicador obligatorio que debe estar el primero, mientras que `T` es el carácter que separa la fecha de la hora y únicamente debe aparecer si se indica una hora. Ninguno de los elementos es obligatorio ni tiene limitación de rango. También se pueden indicar duraciones negativas precediéndolas del signo '-' (si se omite el signo, se tratarán como duraciones positivas). Un valor de tipo *duration* podría ser, por ejemplo, "P1DT2S" ("duración de un día y dos segundos").

También se van contemplar tipos secuenciales. Como ya conocemos de otros lenguajes, las secuencias son un tipo de datos cuyos elementos están ordenados y pueden ser accedidos vía un índice.

- ***list***. Una lista es una colección de elementos del mismo tipo.

- ***tuple***. Una tupla es una colección de elementos que no tienen por qué ser del mismo tipo.

También podríamos considerar el tipo ***string*** como una secuencia de elementos de tipo carácter, pero como no se ha incluido el tipo ***char*** en el sistema, se ha clasificado el tipo ***string*** como primario.

Las declaraciones de variables del catálogo van a estar formadas por dos tipos de elementos:

Variable Representa una variable que será utilizada en la plantilla de un mensaje. Contiene dos atributos:

Name Cadena de caracteres que identifica la «variable» de manera única para un determinado mensaje.

Type Cadena de caracteres que indica el tipo de la «variable». Puede ser uno de los tipos primitivos o secuencia predefinidos, o bien el nombre de un tipo («typedef») definido por el usuario.

Typedef Se trata de un tipo de variable "especial". Sirve para definir nuevos tipos a partir de los básicos u otros elementos «typedef»: renombrar un tipo primitivo, declarar el tipo de los elementos de una lista/tupla que será usado en una variable, definir variables escalares cualesquiera basados en tipos primitivos con restricciones, etc. En realidad los «typedef» son declaraciones de tipos de «variable», no variables propiamente dichas, por lo que no aparecen nunca dentro de la sección «template». Poseen dos atributos obligatorios:

Name Cadena de caracteres que identifica el «typedef» de manera única para un determinado mensaje.

Type Cadena de caracteres que indica el tipo del «typedef». Sólo puede tomar el valor de uno de los tipos primitivos o secuencia predefinidos.

También se permite definir otra serie de atributos:

element Este atributo es obligatorio si el atributo `type` tiene valor ***list*** o ***tuple***. En tal caso, representa el tipo de los elementos. En el caso de las tuplas podríamos necesitar una lista de tipos, por lo que el valor de `element` es una lista ordenada de cadenas separadas por coma.

min Este atributo tiene un significado u otro en función del tipo al que se aplique. Aplicado a tipos numéricos, representa el límite inferior inclusivo del espacio de valores del tipo. En el caso de que se aplique a un tipo cadena, indica la longitud mínima que ésta ha de tener. Sin embargo, si es un tipo ***list***, `min` indica el número mínimo de elementos que puede contener la lista.

max De forma análoga a `min`, este atributo representa el límite superior inclusivo del espacio de valores del tipo de datos al que se aplique cuando éste es numérico, la longitud máxima si es un tipo ***string*** o derivado y el número máximo de elementos cuando se trate de un tipo ***list***.

values Restringe el espacio de valores de un determinado tipo al conjunto de valores especificados.

pattern Restringe el espacio de valores de un determinado tipo, restringiendo el espacio léxico a literales que siguen un determinado patrón. El valor de `pattern` debe ser una expresión regular.

fractionDigits Controla el tamaño de la mínima diferencia entre valores de un tipo (de los derivados del tipo decimal de XML Schema), restringiéndolo a números que se pueden expresar como $i \times 10^n$ donde i y n son enteros tal que $0 \leq n \leq totalDigits$. El valor de `fractionDigits` debe ser un entero no negativo.

totalDigits Controla el máximo número de valores de un tipo (de los derivados del tipo decimal de XML Schema), restringiéndolo a números que se pueden expresar como $i \times 10^n$ donde i y n son enteros tal que $|i| < 10^{totalDigits}$ y $0 \leq n \leq totalDigits$. El valor de `totalDigits` debe ser un entero positivo.

Listado 4.4: Ejemplos de declaraciones

```
<mes:typedef name="boolean" type="string" values="false,true"/>
<mes:typedef name="natural" type="int" min="0"/>
<mes:typedef name="price" type="float" fractionDigits="2" totalDigits="13"/>
<mes:typedef name="D1" type="date" min="2011−01−01"/>
<mes:typedef name="T1" type="tuple" element="string,boolean"/>
<mes:typedef name="L1" type="list" element="string,boolean" min="5" max="5"/>
<mes:variable name="request" type="string"/>
<mes:variable name="boActivo" type="boolean"/>
```

4.5.3. Plantillas

Las plantillas de los mensajes van a incluirse en el catálogo en el interior de un bloque CDATA. Esto se debe a que el código de las plantillas no es puramente XML sino que va a contener también código VTL (Velocity Template Language).

Apache Velocity permite generar contenido dinámico en una plantilla mediante referencias. Existen tres tipos de referencias en VTL: variables, propiedades y métodos. Por el momento, en las plantillas no vamos a utilizar ni

propiedades, ni métodos [7]. Sí que van a aparecer variables para hacer posible la parametrización de las plantillas.

La notación breve [8] de una variable esta compuesta por un signo "$" inicial seguido de un identificador. Un identificador VTL debe comenzar con un carácter alfabético (a .. z, A .. Z) y seguir con caracteres alfabéticos, numéricos (0 .. 9) o guiones ("-", "_"). Hay que tener en cuenta que Velocity distingue entre mayúsculas y minúsculas.

Las variables pueden ser escalares o tipo secuencia. Las listas son definidas con el operador [..] y son accesibles usando los métodos definidos en la clase ARRAYLIST.

Listado 4.5: Ejemplos de variables Velocity

```
# Esto es un comentario Velocity
$cont # variable escalar
$notas.get(0) # accede al primer elemento de la variable lista "notas"
```

Simplemente con las referencias no podemos controlar la apariencia y el contenido de los mensajes que se generarán a partir de las plantillas. También necesitamos directivas, es decir, elementos de *script* fáciles de usar que permiten manipular de manera creativa la salida del código Java. Las directivas Apache Velocity que podemos encontrarnos en una plantilla son las siguientes:

- `#set`: se utiliza para establecer el valor de una referencia. El valor se puede asignar a una referencia de variable o una referencia de propiedad, siempre entre paréntesis, como se muestra a continuación:

[7] Las propiedades y métodos VTL tienen el mismo significado que los atributos y métodos, respectivamente, en la programación orientada a objetos tradicional.

[8] Existe también la notación formal que consiste en introducir entre llaves todo el identificador de la variable, pero esta notación no se utiliza por el momento en las plantillas generadas por *ServiceAnalyzer*. La notación formal es útil cuando las referencias están ubicadas directamente al lado del texto dentro de una plantilla y hay que indicar dónde acaba la variable y donde empieza el texto.

Listado 4.6: Ejemplos de asignaciones Velocity

```
#set( $cont = 0 )
#set( $cont = cont + 1 )
#set( $notas = ['suspenso', 'aprobado', 'notable', 'sobresaliente'] )
```

- `#if`: permite que se incluya texto dentro de la plantilla generada, con la condición de que el enunciado condicional evalúe a verdadero. Un elemento `#elseif` o `#else` puede utilizarse junto con una sentencia `#if` para indicar condiciones adicionales.

Listado 4.7: Ejemplo de condicional en Velocity

```
#if ($cont > 0)
        $total = "$cont"
#end
```

- `#foreach`: esta directiva permite la construcción de bucles.

Listado 4.8: Ejemplo de bucle en Velocity

```
#foreach ($url in $UrlList)
        <url>$url</url>
#end
```

En las plantillas que genera *ServiceAnalyzer* podemos encontrar asimismo la cadena '`retnull`' que devuelve `null`.

En el anexo B se profundiza en el uso de plantillas Velocity en los ficheros de casos de prueba para `BPELUnit`.

4.5.4. Reglas de reescritura

Podemos pensar en *ServiceAnalyzer* como una especie de traductor o sistema de reescritura. Cuando el sistema analiza la construcción XML Schema

asociada a un mensaje, aplica a dicha construcción una función de transformación para obtener la plantilla correspondiente. Es decir, si encuentra una construcción x que siga un determinado patrón, el resultado será $T(x)$, donde T es la función que modela a la transformación y cuyo resultado es la plantilla asociada a x.

Por ello, se han definido una serie de reglas de alto nivel para la transformación, reglas que al fin y al cabo describen los requisitos de lo que tiene que hacer el programa. De manera análoga, también se han descrito una serie de reglas para la generación de las declaraciones, que se obtienen a través de la función de transformación $D(X)$.

4.5.4.1. Reglas para la generación de plantillas

En realidad, en las plantillas podemos considerar dos cosas: la construcción XML Schema cuya plantilla se quiere generar y la variable principal asociada a esa plantilla, es decir, la que debería incluirse en la fuente de datos. Decimos "principal" porque pueden existir otras variables auxiliares que contribuyan a la generación del los mensajes, pero que no son las que se deben sustituir por valores diferentes en cada caso de prueba.

Por ello, vamos a considerar la función $T(e, v)$ donde e es la construcción XML Schema del nivel analizado y v la variable asociada a ese nivel.

Por otro lado, el punto de partida va a ser la construcción XML Schema a la que hace referencia el atributo `type` o el `element`, según el caso, de la parte de un mensaje extraído de un documento WSDL. Si se parte de un atributo `type`, se va a considerar que la construcción XML Schema es la compuesta por un elemento «xsd:element» cuyos atributos `name` y `type` tienen los mismo valores que los del elemento «part» asociado. Al elemento raíz de la construcción se le asigna una variable de la fuente de datos.

Pues bien, ya podemos pasar a definir las reglas:

- Conversión de un tipo XSD nativo: una construcción «xsd:nombretipo» en la que "nombretipo" es el nombre de uno de los tipos nativos de XML Schema (primitivo o derivado, como por ejemplo *xsd:string* o *xsd:integer*) se transforma en la plantilla asociada a su variable. Es el caso base.

Entrada:	Salida:
xsd:nombretipo	$v

- Conversión de restricciones: una construcción «xsd:restriction» se transforma en un patrón mediante la transformación del tipo base al que hace referencia.

Entrada:	Salida:
<xsd:restriction base="x"> ... </xsd:restriction>	T(x, v)

- Conversión de listas: una construcción «xsd:list» se transforma en un bucle capaz de generar los elementos de la lista.

Entrada:	Salida:
<xsd:list itemType="x"/>	#foreach($v_i in $v) T(x, v_i) #end

- Conversión de secuencias: una construcción «xsd:sequence» se transforma a un patrón mediante una transformación secuencial de sus elementos.

Entrada:	Salida:
<xsd:sequence> <x_1 .../> <x_2 .../> ... <x_n .../> </xsd:sequence>	T(x_1, v.get(0)) T(x_2, v.get(1)) ... T(x_n, v.get(n−1))

- Conversión de repeticiones: una construcción con los atributos «minOccurs» y/o «maxOccurs» definidos se transforma en un bucle capaz de generar las repeticiones que se derivan.

Entrada:	Salida:

<x minOccurs="..." maxOccurs="..."/>	#foreach($v_i in $v)
	T(x, v_i)
	#end

- Conversión de elementos: la transformación de una construcción «xsd:element» dependerá de la construcción tipo al que haga referencia (la determinada por el atributo `type`). Así, se distinguen dos casos:

  - Tipos simples: una construcción «xsd:simpleType» se transforma a un elemento XML con el *QName* del «xsd:element» asociado. El contenido del nuevo elemento XML es el resultado de transformar el tipo referenciado en el atributo `type`.

Entrada:	Salida:
<xsd:element name="e" type="t" .../> <xsd:simpleType name="t" ...> <x .../> </xsd:simpleType>	<ns:e> T(x, v) </ns:e>

  - Tipos complejos: una construcción «xsd:complexType» se transforma a un elemento XML con el *QName* del «xsd:element» asociado.

El contenido del nuevo elemento XML es el resultado de transformar el tipo en el atributo `type`. La transformación se hace de forma secuencial, atributos por separado (si los hubiese).

Entrada:

```
<xsd:element name="e" type="t" .../>
<xsd:complexType name="t" ...>
        <x .../>
        <xsd:attribute name="a_1" .../>
        <xsd:attribute name="a_2" .../>
        ...
        <attribute name="a_n" .../>
</xsd:complexType>
```

Salida:

```
<ns:e #if(v_ea1.get(i_ea1) != 'retnul') a_1="v_ea1.get(i_ea1)" #end #set(i_ea1
    = i_ea1 + 1)
                        #if(v_ea2.get(i_ea2) != 'retnul') a_2="v_ea2.get(i_ea2)"
                            #end #set(i_ea2 = i_ea2 + 1)
                        ...
                        #if(v_ean.get(i_ean) != 'retnul') a_n="v_ean.get(i_ean)"
                            #end #set(i_ean = i_ean + 1)
>
        T(x, v)
</ns:e>
```

4.5.4.2. Reglas para la generación de declaraciones

El punto de partida es el mismo que para el algoritmo de generación de plantillas. Sin embargo, no hace falta un parámetro adicional.

- Conversión de un tipo XSD nativo: una construcción «xsd:nombretipo» en la que "nombretipo" es el nombre de uno de los tipos nativos de XML Schema se transforma en el nombre del tipo del sistema definido por

ServiceAnalyzer o bien, en un elemento «typedef» con las restricciones correspondientes. La traducción de tipos entre los tipos XML Schema primitivos y los de nuestro sistema se detalla en la tabla 4.3 (página 170).

Entrada:	Salida I:
xsd:nombretipo	tipo
	Salida II:
	typedef

- Conversión de restricciones: una construcción «xsd:restriction» se transforma en un «typedef» en el que el tipo es el resultado de transformar el tipo base de dicha construcción y cuyos atributos dependen de la traducción de las facetas que contenga.

Entrada:	Salida:
<xsd:restriction base="x"> <r_1 .../> <r_2 .../> ... <r_n .../> </xsd:restriction>	<typedef type="D(X)" D(r_1) D(r_2) ... D(rn) />

  Sea cada r_i de la forma:

```
<xs:faceta value="X"/>
```

  la traducción a las restricciones propias de los «typedef» se detalla en la tabla 4.2.

- Conversión de listas: una construcción «xsd:list» se transforma en un elemento «typedef» de tipo ***list*** cuyos elementos serán del tipo resultado de transformar el tipo apuntado por `itemType`.

Faceta XML Schema	Restricción Typedef	Valor
maxInclusive	max	x
minInclusive	min	x
maxExclusive	max	$x-1$
minExclusive	min	$x+1$
maxLength	max	x
minLength	min	x
length	min max	x
fractionDigits	fractionDigits	x
totalDigits	totalDigits	x
enumeration	values	x

Tabla 4.2: Equivalencias entre restricciones

Entrada:

<xsd:list itemType="x"/>

Salida:

<typedef type"list" element="D(x)"/>

- Conversión de secuencias: una construcción «xsd:sequence» se transforma en un elemento «typedef» de tipo ***tuple*** cuyos elementos serán del tipo resultado de la transformación secuencial de sus elementos.

Entrada:

<xsd:sequence>
<x_1 .../>
<x_2 .../>
...
<x_n .../>
</xsd:sequence>

Salida:

<typedef type="tuple" element="D(x_1),
D(x_2),...,D(x_n)"/>

- Conversión de repeticiones: una construcción con los atributos «minOccurs» y/o «maxOccurs» definidos se transforma en un elemento «typedef» de tipo ***list*** con el mínimo y/o máximo indicado y cuyos elementos serán del tipo resultado de transformar el contenedor de dichos atributos.

Entrada:	Salida:
<x minOccurs=”m” maxOccurs=”M”/>	<typedef type”list” element=”D(x)” min=” m” max=”M”/>

- Conversión de elementos: la transformación de una construcción «xsd:element» dependerá de la construcción tipo al que haga referencia (la determinada por el atributo `type`). Así, se distinguen dos casos:

  - Tipos simples: una construcción «xsd:simpleType» se transforma en un patrón mediante la transformación de su contenido.

    Entrada:

    ```
    <xsd:element name=”e” type=”t” .../>
    <xsd:simpleType name=”t” ...>
            <x .../>
    </xsd:simpleType>
    ```

    Salida:

    ```
    D(x)
    ```

  - Tipos complejos: una construcción «xsd:complexType» se transforma a un patrón mediante la transformación secuencial del contenido (atributos por separado, si los hubiese).

    Entrada:

    ```
    <xsd:element name=”e” type=”t” .../>
    <xsd:complexType name=”t” ...>
            <x .../>
            <xsd:attribute name=”a_1” type=”ta_1”/>
            <xsd:attribute name=”a_2” type=”ta_2”/>
            ...
            <attribute name=”a_n” type=”ta_n”/>
    </xsd:complexType>
    ```

Salida:

<typedef name="t1" type="list" element="D(ta_1)"/>
<variable name="v_ea1" type="t1"/>
<variable name="i_ea1" type="int"/>
<typedef name="t2" type="list" element="D(ta_2)"/>
<variable name="v_ea2" type="t2"/>
<variable name="i_ea2" type="int"/>
...
<typedef name="tn" type="list" element="D(ta_n)"/>
<variable name="v_ean" type="tn"/>
<variable name="i_ean" type="int"/>
D(x)

Además, en ambos casos, si el elemento es el elemento raíz de la construcción inicial, entonces habrá que añadir a los resultados anteriores:
Salida:

<variable name="e" type="D(x)"/>

Cabe comentar el caso de los atributos. Como se puede observar en las reglas, se genera una lista del tipo del atributo. Esto es debido a que el atributo podría no aparecer en un mensaje concreto o podría aparecer más de una vez si el elemento que lo contiene se repite. De ahí la complicada estructura de la plantilla con un contador que lleva la cuenta de las veces que ha aparecido el atributo y un condicional que comprueba si el atributo que toca generar debe aparecer o no y con qué valor.

4.5.5. Arquitectura del sistema

Es el momento de centrar la atención en los aspectos relativos a la estructura del programa.

En las figuras 4.5 y 4.6 podemos ver una representación simplificada de esta estructura. En realidad podíamos tener un único diagrama, pero se ha partido en dos por claridad. Algunas clases se hallan duplicadas en ambos diagramas e incluso en el mismo diagrama. Para evitar confusiones, las duplicaciones ocultan los atributos y se hallan en color azul. También por claridad, se han omitido las multiplicidades unitarias en las asociaciones.

En el caso del diagrama de la figura 4.6, se ha mostrado con más o menos detalle los componentes de cada clase según se ha considerado conveniente para entender la estructura representada. Asimismo, se han destacado las interfaces en naranja, también por motivos de claridad.

SERVICEANALYZER es la clase encargada de generar el catálogo de mensajes y sacarlo por pantalla, aunque esto sólo se hace cuando así lo solicita el usuario a través de la orden SERVICEANALYZER.

El manejo de la línea de órdenes y la interfaz de usuario está separada, basándonos en el patrón Comando, en las clases APPLICATION y SERVICEANALYZERCOMMAND.

El componente SCHEMASANALYZER facilita la lectura de los ficheros XML Schema necesarios para la traducción de tipos. Son las clases de *WSDL2XSDTree* las encargadas de organizar todos los XML Schema jerárquicamente antes de pasar por el analizador.

Por otro lado, el PARSER es el componente del que SERVICEANALYZER se sirve para, dada una construcción XML Schema asociada a un mensaje, realizar el análisis cuyo resultado determinará la manera de construir la plantilla y las declaraciones correspondientes. En cierto modo, es la clase que implementa las reglas de reescritura ya explicadas.

Las clases con las que se relaciona el PARSER conforman la estructura intermedia que se ha diseñado para separar la generación de plantillas de la ge-

neración de declaraciones. Esta estructura es la que se representa en la figura de la página 122 y la explicamos con más detalle en el siguiente apartado.

En cuanto a *WSDL2XSDTree*, la clase SINGLEWSDLTRANSFORMER es encargada de transformar un documento WSDL en un documento XSD en el que se mantengan los «XSD:import», se transformen los «WSDL:import» en «XSD:import» y se incluyan los XML Schema incrustados en sección «types». Por su lado, WSDLTREETRANSFORMER llama recursivamente a la clase anterior, la primera vez con el documento WSDL de entrada y posteriormente, con los documentos WSDL importados por éste, dando lugar al árbol XSD de salida.

4.5.5.1. AST

Para mejorar la reusabilidad y facilitar las pruebas, evitando acoplamientos innecesariamente complejos que dificultaran el desarrollo de la aplicación, se ha decidido crear un árbol sintáctico intermedio a partir del cual:

- Crear la plantilla Velocity correspondiente a un mensaje.
- Generar las declaraciones de tipo de las variables empleadas en la plantilla.

Para ello se han definido una serie de nodos que contendrán la información necesaria para poder, en un paso posterior, generar la plantilla o las declaraciones, según corresponda.

Un AST puede contener los siguientes tipos de nodo:

Elemento (ELEMNODE) Se crea a partir de un «xsd:element» y almacena el *QName* del mismo, los atributos (si posee alguno) y el nodo que representa su cuerpo o contenido.

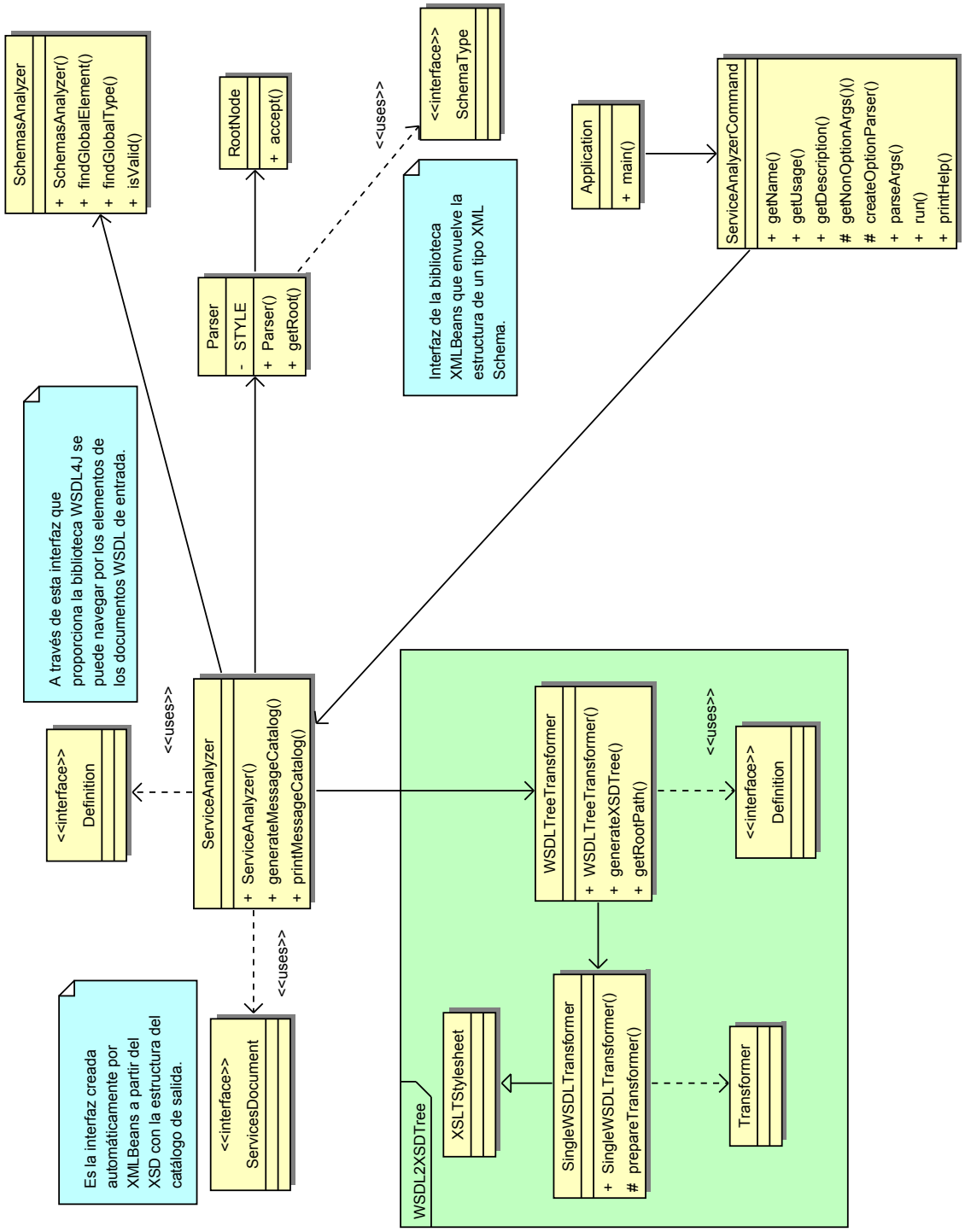

Figura 4.5: Diagrama de clases de ServiceAnalyzer (I)

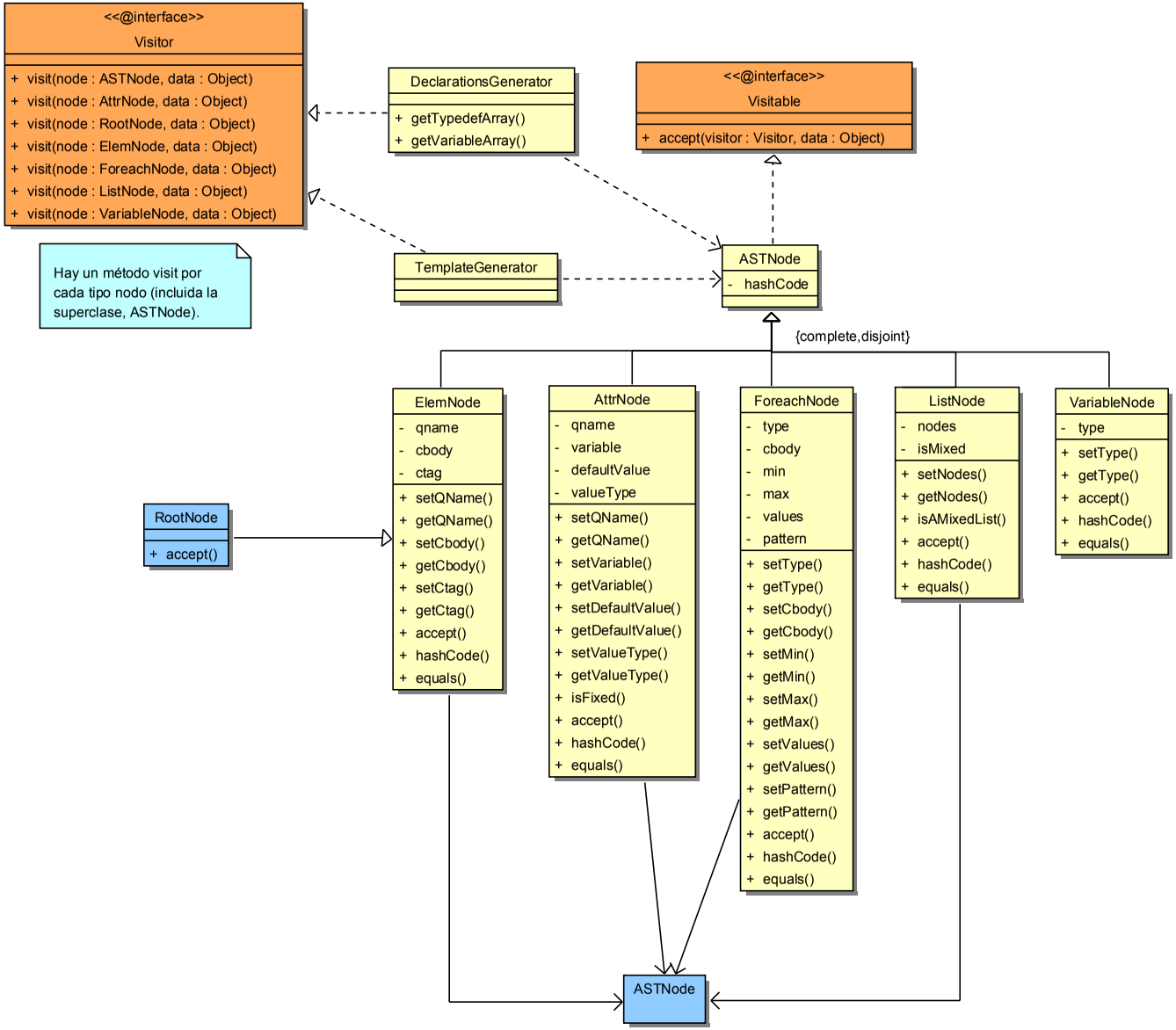

Figura 4.6: Diagrama de clases de ServiceAnalyzer (II)

Raíz (RootNode) Se trata del nodo elemento del que parten todas las ramas del árbol correspondiente a un mensaje. Por tanto, siempre hay uno, y sólo uno, por árbol.

Atributo (AttrNode) Representa el atributo de un elemento. Dicho atributo puede tener un valor fijo o un valor por defecto, o bien, ninguna restricción sobre el valor más que su tipo. Tiene una variable asociada que representa el valor del atributo en la plantilla.

Variable (VariableNode) Representa una variable en base a la cual se logra que las plantillas sean paramétricas. Guarda una representación del tipo asociado a dicha variable, en el sistema de tipos definido para *ServiceAnalyzer*.

Bucle (ForeachNode) Se crea cuando se lee un componente XSD que puede repetirse o son más de uno los valores que toma el componente (lista simple). El número de repeticiones puede tener un mínimo y/o un máximo. Incluso, si se trata de repetición de valores, éstos pueden seguir un patrón determinado o estar restringidos a un conjunto concreto, información que también es necesario almacenar.

Lista (ListNode) Se trata de una agrupación lógica de nodos. Esta agrupación se da en dos casos: cuando un elemento tiene más de un atributo asociado (lista de atributos) y cuando hay una secuencia de elementos (lista de elementos).

Un árbol no es más que un nodo raíz que se encarga de apuntar al resto de nodos.

Podemos decir que cuando se genera un mensaje del catálogo a partir del AST correspondiente, los nombres de las variables se propagan hacia abajo en el árbol, mientras que los tipos de las variables se propagan hacia arriba, algo

similar al concepto de atributos heredados y atributos sintetizados (respectivamente) en un traductor.

Veamos un ejemplo. Dada la siguiente construcción XML Schema del listado 4.9 el árbol correspondiente es el representado en la figura 4.7. Los nodos del árbol se representan en azul, los atributos de cada nodo en amarillo (gris si tienen valor `null`) y en las notas se indica el valor de estos atributos. Por otro lado, por cada nodo el AST, a su izquierda hemos indicado el nombre de la variable que propaga hacia abajo, y a la derecha, el nombre del tipo que propagan hacia arriba.

Listado 4.9: Código XML Schema del ejemplo

```
1 <xsd:element name=”aspirantes”>
2 <xsd:complexType>
3         <xsd:sequence minOccurs=”0” maxOccurs=”50”>
4                 <xsd:element name=”nombre” type=”xsd:string”/>
5                 <xsd:element name=”admitido” type=”xsd:boolean”/>
6         </xsd:sequence>
7         <xsd:attribute name=”total” type=”xsd:int”/>
8 </xsd:complexType>
```

La plantilla resultante sería:

Listado 4.10: Código de la plantilla del ejemplo

```
1 #set($aspirantes_total_i = 0)
2 <ns1:aspirantes xmlns:ns1=”www.ejemplo.com”
3         #if($aspirantes_total.get($aspirantes_total_i) != ’retnull’) total = ”$aspirantes_total.
              get($aspirantes_total_i)” #end     #set($aspirantes_total_i = $aspirantes_total_i
              + 1)>
4 #foreach($V1 in $aspirantes)
5 <ns1:nombre>$V1.get(0)</ns1:nombre>
6 <ns1:admitido>$V1.get(1)</ns1:admitido>
7 #end
8 </ns1:aspirantes>
```

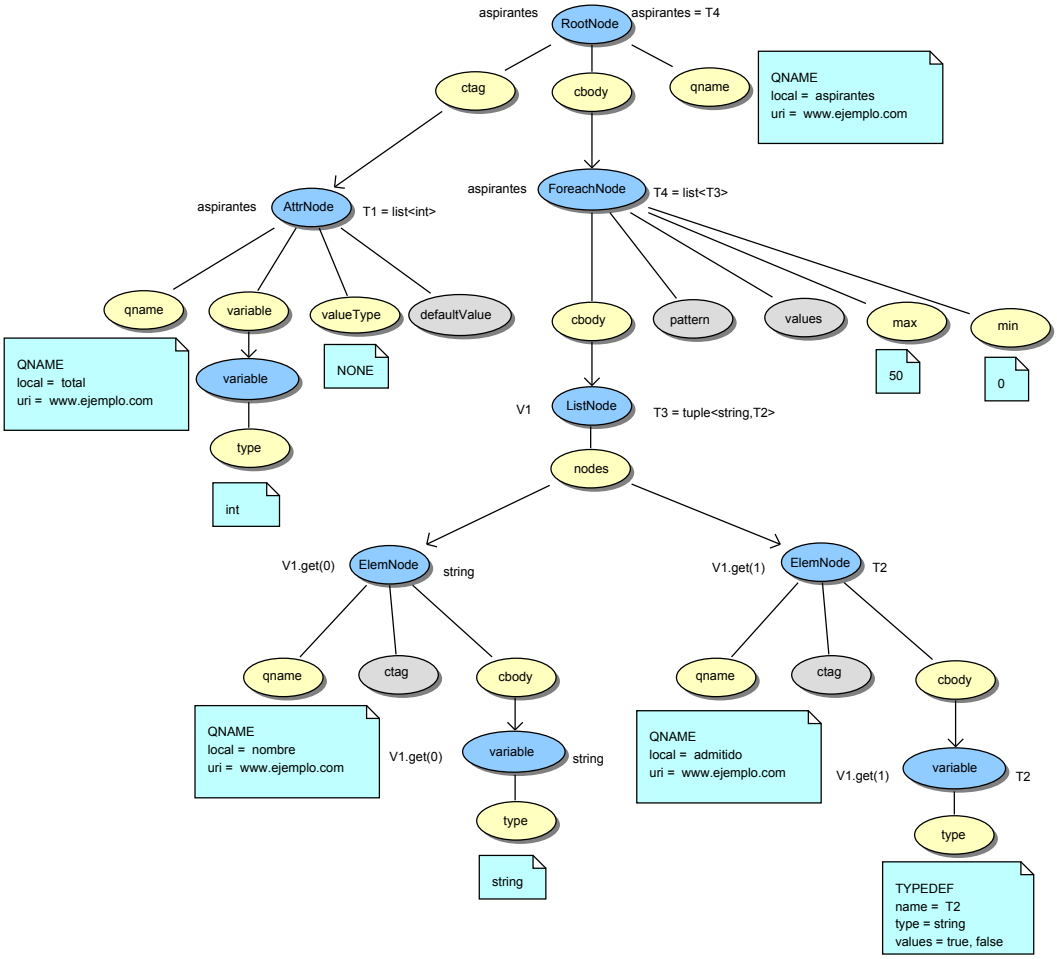

Figura 4.7: Representación ejemplo de un AST

Las declaraciones generadas serían:

Listado 4.11: Código de las declaraciones del ejemplo

```
1 <typedef name="T1" type="list" element="int"/>
2 <typedef name="T2" type="string" values="[true:false]"/>
3 <typedef name="T3" type="tuple" element="string,␣T2"/>
4 <typedef name="T4" type="list" element="T3" min="0" max="50"/>
```

```
5 <variable name="aspirantes_total" type="T1"/>
6 <variable name="aspirantes_total_i" type="int"/>
7 <variable name="aspirantes" type="T4"/>
```

Para el diseño de la arquitectura de esta parte (la más significativa) del software se ha empleado el patrón Visitante, como queda reflejado en la figura 4.6 (página 122).

4.5.5.2. Patrón Visitante

Se trata de un patrón de comportamiento [27, 55] cuyo propósito es, dada una estructura de objetos, permitir la definición de nuevas operaciones que actúen sobre ella sin cambiar las clases de estos elementos. Como consecuencia, este patrón encapsula comportamiento que de otro modo estaría distribuido en varias clases.

Participantes

- VISITANTE: interfaz que declara una operación *Visitar* para cada clase de operación ELEMENTO CONCRETO de la estructura de objetos. El VISITANTE puede acceder al elemento directamente a través de su interfaz particular.

- VISITANTE CONCRETO: implementa cada operación declarada por la interfaz VISITANTE. Proporciona el contexto para el algoritmo y almacena el estado local (normalmente acumula los resultados durante el recorrido de la estructura).

- ELEMENTO: define una operación *Aceptar* que toma un VISITANTE como argumento.

- ELEMENTO CONCRETO: implementa una operación *Aceptar* que toma un VISITANTE como argumento.
- ESTRUCTURA DE OBJETOS: puede ser una composición o colección de objetos (una lista o un conjunto, por ejemplo), en la que sus elementos se pueden enumerar. Proporciona una interfaz de alto nivel para que los visitantes visiten sus elementos.

Un cliente que usa el patrón Visitante debe crear un objeto VISITANTE CONCRETO y a continuación, recorrer la estructura, visitando cada objeto con el visitante. Cada vez que se visita un elemento, éste llama a la operación del VISITANTE que se corresponde con su clase. El elemento se pasa a sí mismo como argumento de la operación para permitir al VISITANTE acceder a su estado, si es necesario. Para recorrer la estructura, cada ELEMENTO CONCRETO llama al método *Aceptar* de sus descendientes con el VISITANTE CONCRETO.

Conceptualmente, la estructura del código es la siguiente:

Listado 4.12: Estructura del código del patrón Visitante

```
1  class Node {
2  ...
3          void accept(Visitante v) {
4                  for each child of this node {
5                          child.accept(v);
6                  }
7                  v.visit(this);
8          }
9  }
10
11 class Visitor {
12 ...
13         void visit(Node n) {
14                 perform work on n
```

```
15          }
16  }
```

Características

1. Facilita la definición de nuevas operaciones: añadir una nueva operación conlleva únicamente añadir un nuevo VISITANTE CONCRETO.

2. Agrupa operaciones relacionadas y separa las que no lo están: el comportamiento similar no está desperdigado por las clases que definen la estructura de objetos sino que está localizado en un visitante. Las no relacionadas, se dividen en sus propias subclases del visitante. Esto simplifica tanto las clases que definen los elementos, como los algoritmos definidos por los visitantes. Cualquier estructura de datos específica de un algoritmo puede estar oculta en el visitante.

3. Añadir nuevas clases ELEMENTO CONCRETO es costoso: cada ELEMENTO CONCRETO nuevo da lugar a una nueva operación abstracta del VISITANTE y a su correspondiente implementación en cada clase VISITANTE CONCRETO.

4. Permite atravesar varias jerarquías de clases: a diferencia de un iterador, este patrón puede visitar objetos que no están relacionados por un padre común.

5. Permite acumular el estado: los visitantes pueden acumular estado a medida que van visitando cada elemento de la estructura de objetos, en vez de pasarlo como argumento o usar variables globales.

6. Rompe la encapsulación: el enfoque de este patrón arquitectónico asume que la interfaz de ELEMENTO CONCRETO es lo suficientemente potente

como para que los visitantes hagan su trabajo. Como consecuencia, el patrón suele obligarnos a proporcionar operaciones públicas que accedan al estado interno de un elemento, lo que puede comprometer su encapsulamiento.

Implementación

La clave del patrón visitante es el *doble despacho*. Ésta es la técnica que permite añadir operaciones a las clases sin tener que modificarlas. Dada una operación, se dice que es de doble despacho si su ejecución depende de la clase de petición y de los tipos de dos receptores. En el caso del patrón Visitante, la petición es el *Aceptar* y los receptores el VISITANTE y el ELEMENTO.

Un VISITANTE debe recorrer cada ELEMENTO de la estructura de objetos. La responsabilidad del recorrido puede recaer sobre:

- La propia estructura de objetos: una colección iterará sobre sus elementos simplemente invocando a la operación *Aceptar* de cada uno de ellos. Esta es la opción más común.

- Un objeto iterador aparte: se puede hacer uso de los iteradores que ofrecen algunos lenguajes como C++.

- El Visitante: poner el algoritmo de recorrido en el visitante tiene la finalidad de implementar un recorrido especialmente complejo que dependa de los resultados de las operaciones de la estructura de objetos.

En nuestro caso, la estructura de objetos es el AST mediante el que representamos los tipos XML Schema que se necesitan para la generación de las plantillas y sus respectivas declaraciones de variables.

Vamos a tener, por tanto, dos clases VISITANTE CONCRETO:

- *DeclarationsGenerator*: opera sobre los nodos del árbol para determinar las declaraciones de variables del mensaje que se está analizando.
- *TemplateGenerator*: opera sobre los nodos del árbol para construir la plantilla del mensaje correspondiente.

Sin embargo, la estructura de implementación descrita anteriormente no se ajusta a nuestras necesidades. El motivo es que no considera el caso de que un visitante de un elemento dependa de los resultados de la visita de los hijos, es decir, no ofrece modo alguno de hacer que una llamada de *Visitar* se comunique con otra.

Podemos encontrar dos posibles soluciones a este problema. La primera consiste guardar la información en una estructura de datos separada (por ejemplo, una pila) que pueda ser leída y escrita. Esto deja limpios a los visitantes y a los aceptantes, pero puede ser difícil ver cómo los datos fluyen entre las llamadas. La segunda, propone desplazar parte del trabajo para el propio visitante, que es la opción que se ha tomado y que puede esquematizarse como:

Listado 4.13: Patrón Visitante con el recorrido a cargo del *Visitante*

```
1  class Node {
2  ...
3          void accept(Visitor v) {
4                  v.visit(this);
5          }
6  }
7
8  class Visitor {
9  ...
10         void visit(Node n) {
11                 for each child of this node {
12                         child.accept(v);
```

```
13                  }
14                  perform work on n
15          }
16  }
```

Esta solución presenta varios problemas. Por un lado, existen muchos visitantes, por lo que el código de búsqueda se repite varias veces en lugar de aparecer sólo una vez (ya que sólo hay un aceptante). Por otro, el aceptante no hace ya nada más. El visitante está haciendo esencialmente una búsqueda con detalle por sí sólo. Sin embargo, esta solución hace que el flujo de información fluya más claro, por ello, se ha hecho uso del patrón a pesar de todo.

Es resumen, el patrón Visitante típicamente tiene dos ventajas: reutilizar el recorrido y seleccionar el método visitante en base al tipo real del nodo. Al controlar el recorrido, perdemos la primera ventaja, pero mantenemos la segunda, obteniendo un código mucho más ordenado y mantenible.

4.6. Codificación

Para este proyecto se ha seguido un enfoque orientado a objetos, que es el que Java sigue. Nos hemos decantado por este enfoque por las numerosas ventajas de la orientación a objetos, entre ellas:

- Se aproxima más a la forma de pensar de las personas, lo que facilita la comprensión.
- Mejora la intercomunicación entre expertos en el dominio del problema, usuarios, analistas y diseñadores.
- Proporciona abstracción, ya que en cada momento se consideran sólo los elementos que nos interesan descartando los demás.

- Aumenta la consistencia interna al tratar los atributos y las operaciones como un todo.
- Permite expresar características comunes sin repetir la información.
- Facilita la reutilización de diseños y códigos.
- Facilita la revisión y modificación de los sistemas desarrollados.
- Produce sistemas más estables y robustos.

El proceso de codificación, puede ser resumido en los siguiente pasos:

1. **Definición de la estructura de la salida del programa.** Como ya se ha mencionado, ServiceAnalyzer debe generar un catálogo con formato XML en el que se recojan todos los posibles mensajes SOAP que se intercambian entre la composición y los *mockups*. Estos mensajes deben estar categorizados para facilitar, posteriormente, su incrustación en un fichero BPTS. Esta categorización no es más que la clasificación de los mensajes dentro del servicio concreto al que pertenezcan, el puerto, la operación y si se trata de un mensaje de solicitud, respuesta o fallo.
2. **Representación en memoria del catálogo de mensajes.** Se va a necesitar convertir la estructura anterior a código Java para tener una representación en memoria con la que trabajar mientras se realiza el análisis de los ficheros WSDL y XML Schema de entrada. Es decir, a medida que se realice el análisis, se irá "rellenando" el catálogo creando objetos en memoria y dándoles los valores obtenidos en este análisis.
3. **Lectura de los ficheros WSDL**. Una vez concretados los pasos anteriores, lo siguiente es establecer los mecanismos mediante los cuales ir leyendo los documentos WSDL con el fin de obtener la información

necesaria para inicializar los elementos del catálogo que conforman el contexto de los mensajes. También se añadirán las plantillas y las declaraciones de cada mensaje a partir de los nombres de tipos o elementos determinados en este paso. Sin embargo, la generación de ambas será tratada aparte por su complejidad.

4. **Lectura y análisis del contenido XML Schema.** Para generar las plantillas y las declaraciones de sus variables, es necesario leer el código XML Schema incrustado o importado en los documentos WSDL, buscar el componente (bien sea un elemento, bien un tipo) correspondiente al mensaje que se está procesando y finalmente, analizar su estructura. Este análisis requiere el máximo nivel de detalles, ya que a partir del mismo, se debe poder generar las plantillas paramétricas, y por consiguiente, las declaraciones de tipos de las variables que hacen posible la parametrización.

5. **Construcción de una representación en memoria del sistema de tipos definido por todos los ficheros WSDL y XML Schema necesarios.** Como ya se ha adelantado, esta representación se hace a través de un sistema de nodos que, jerarquizados, determinan el AST. A partir de éste, ya tenemos toda la información necesaria para proceder a generar las plantillas y las declaraciones.

6. **Generación de plantillas.** Este paso es de suma importancia, pues son las plantillas las que el usuario utilizará para generar sus casos de prueba. Recordemos que otro punto importante es que las plantillas se generan de forma paramétrica, en base a una serie de variables que permiten crear más de un caso de prueba con el mismo tipo de mensaje, pero distinto contenido (valor asignado a las variables).

7. **Generación de declaraciones.** Si además añadimos la declaración de las variables utilizadas en las plantillas, se facilita la automatización del proceso de generación de casos de prueba. Podrá ser otro componente (y no el propio usuario el) el que asigne diferentes valores válidos a las variables.

8. **Conversión del catálogo generado al formato XML.** Una vez se ha rellenado todo el catálogo, sólo queda mostrarlo en el formato requerido para que el usuario puede darle provecho.

A continuación vamos a comentar detalles de implementación que merece la pena destacar de cada uno de estos pasos, sobre todo en lo referente a las tecnologías utilizadas.

4.6.1. Definición de la estructura de la salida del programa

Como hemos dicho, el formato de la salida de *ServiceAnalyzer* es XML, luego necesitamos un esquema. Un esquema para XML es un método de definición de clases de documentos XML, llamados instancias, que deben ser conformes a las restricciones estructurales y de datos especificadas en el esquema asociado a ellos.

Hay dos metalenguajes con los que definir esquemas para XML que destacan por ser ampliamente utilizados y contar con el apoyo del W3C:

XML DTD (Document Type Definition) es un subconjunto de SGML (Standard Generalized Markup Language) DTD y era el lenguaje de esquema estándar de facto para documentos XML hasta que apareció XML Schema. El objetivo de XML DTD es describir qué elementos, atributos, entidades y notaciones pueden usarse en un documento XML, así como su disposición dentro de él.

XML Schema [59] proporciona los mecanismos básicos para declarar los elementos y atributos que aparecen en documentos XML, distinguiendo entre tipos simples y complejos, que juntos constituyen el modelo de contenido. Existe una importante distinción entre *definiciones*, que crean nuevos tipos, y *declaraciones*, que describen la estructura de elementos y atributos dentro de las instancias de un documento XML.

El enfoque de DTD impone limitaciones importantes tanto sobre la capacidad para validar un documento XML como para definir su estructura, lo que ha llevado progresivamente al reemplazo de XML DTD por XML Schema en el mundo XML. A continuación detallamos las razones de este reemplazo:

- XML Schema utiliza sintaxis XML, mientras que XML DTD no. Esto significa que los usuarios de XML Schema no se ven obligados a aprender una nueva sintaxis propietaria y que el lenguaje de esquema es extensible y puede aplicarse de manera inmediata a aplicaciones XML ya existentes que incluyan un analizador XML.

- A diferencia de DTD, XML Schema permite trabajar con *espacios de nombres*. [9]

- XML Schema define un amplio conjunto de tipos primitivos que cubre la mayor parte de los tipos usados en lenguajes de programación general. Además, permite la utilización de tipos definidos por el usuario. En cambio, XML DTD sólo soporta tipos de datos primitivos de XML.

[9] Un espacio de nombres XML es una colección de nombres, identificada por un URI (Uniform Resource Identifier), que pueden utilizarse como nombres de elementos o atributos en un documento XML. El espacio de nombres, declarado explícitamente o por defecto, cualifica los nombres de elemento de manera única y proporciona un medio de diferenciación entre elementos escritos contra esquemas diferentes, evitando así conflictos entre elementos con idéntico nombre.

- El único mecanismo de XML DTD para limitar el dominio de validez de los valores para tipos de datos es la enumeración, mientras que XML Schema ofrece adicionalmente un conjunto de constructores para este propósito (*facets* [10], expresiones regulares, etc.)

- XML Schema soporta la derivación de tipos por extensión y por restricción, capacidad no presente en XML DTD.

Por consiguiente, para la definición de la estructura del catálogo de mensajes se ha empleado XML Schema, por las razones anteriores y porque, como veremos a continuación, hay numerosas herramientas que facilitan el manejo de documentos XML a partir del XSD [11] contra el que se validan.

4.6.2. Representación en memoria del catálogo de mensajes

Por exigencias del grupo de investigación, la implementación del presente PFC se ha realizado en Java.

Java [5] es una plataforma de computación desarrollada por Sun Microsystems (ahora Oracle). Ofrece soluciones extremo a extremo para aplicaciones en red. Proporciona un lenguaje de programación independiente de la plataforma, lo cual hace el código portable permitiendo que una misma aplicación funcione en diferentes sistemas y dispositivos. Las ventajas de la tecnología Java en términos de portabilidad y extensibilidad la posicionan como una adecuada elección en el desarrollo de aplicaciones que intercambian y procesan información.

Mientras Java será nuestra herramienta de trabajo, el objetivo de *ServiceAnalyzer* es construir el catálogo de mensajes en formato XML.

[10] Las facetas son todas aquellas restricciones que sufren los elementos, valores o atributos XML.

[11] XSD es la extensión de los documentos XML Schema.

Hay varias formas de trabajar con documentos XML. Una de ellas es navegar por el contenido XML con DOM (Document Object Model) [12] , SAX (Simple API for XML) [13] o StAX (Streaming API for XML), que supone una alternativa intermedia entre las dos anteriores. Se trata de una solución primitiva y tediosa ya que requiere entender por completo el modelo de objetos.

Una mejor solución es utilizar herramientas que automaticen la correspondencia de los elementos XML con objetos Java, más conocidas como *XML marshalling/binding tools*.

El vínculo de datos (en inglés, *data binding*) es el proceso de correspondencia de componentes desde un determinado formato de datos a una representación específica para un lenguaje de programación dado. En nuestro caso, XML será el formato origen de los datos y Java, el lenguaje destino al cual se traducirán dichos datos.

La figura 4.8 muestra cómo ocurre este proceso. Un esquema XML se pasa como entrada a un compilador encargado de generar un conjunto de clases Java. De este modo, cualquier documento XML que siga el esquema XML origen puede ser convertido en objetos Java instanciados a partir de las clases generadas. Las aplicaciones pueden así procesar tales objetos con la ventaja de no tener que ceñirse al formato en el que se habían almacenado los datos inicialmente. De esta forma, directamente usamos objetos con sus métodos de acceso y mutación, lo cual puede resultar muy cómodo.

El término *serialización* (en inglés, *marshalling*) significa crear un documento XML de un árbol de contenido. Por consiguiente, el término *deseriali-*

[12]Según, W3C, DOM es una plataforma e interfaz de lenguaje neutro que permite acceder y actualizar dinámicamente el contenido, la estructura y el estilo de documentos. Para ello, genera un árbol jerárquico en memoria del documento, pudiendo agregar o eliminar nodos en el mismo. Contiene interfaces especializadas dedicadas al tratamiento de documentos XML y HTML (Hyper Text Markup Language).

[13]SAX permite procesar información XML conforme ésta sea presentada (secuencialmente o evento por evento). Esto hace que ocupe menos memoria y sea más rápida que DOM, pero impide manipular información una vez procesada.

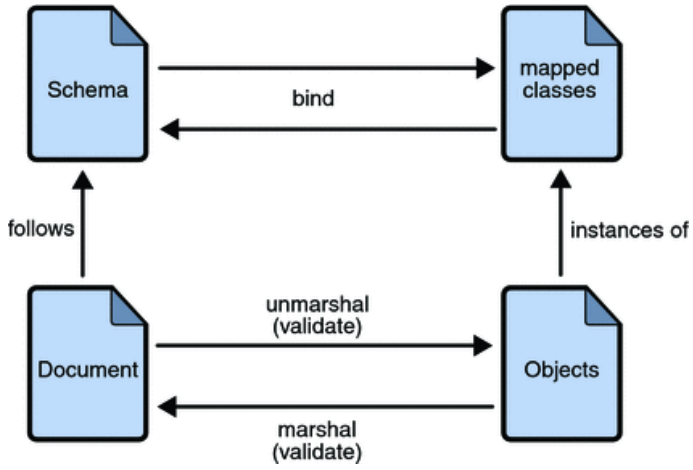

Figura 4.8: Proceso de vinculación de datos y serialización/deserialización

zación significa crear un árbol de contenido de un documento XML.

Actualmente hay disponibles diversas herramientas que realizan automáticamente el *data binding* entre XML y Java. Entre ellas, podríamos destacar las siguientes por ser las más extendidas: `JAXB (Java Architecture for XML Binding)`, `XMLBeans`, `JDOM` o `Castor`.

Estas tecnologías son muy similares. Para nuestro cometido, se ha elegido XMLBeans [20], en primer lugar, porque esta herramienta ya se había empleado en otros trabajos del grupo UCASE con un buen resultado, por lo que no nos adentramos en el estudio y comparación de las otras opciones. Además, presenta las siguientes ventajas:

- Mientras `JAXB`, por ejemplo, provee soporte únicamente para un subconjunto de la especificación XML Schema, `XMLBeans` la soporta al completo.

- `XMLBeans` almacena en memoria la información resultante del análisis sintáctico del XSD de entrada en formato XML, con lo cual es capaz de

reducir la sobrecarga provocada por las actividades de «serializado» y «deserializado».

- Agiliza el acceso a documentos XML y hace más amigable su manejo.

El proyecto `Maven` en que se encuentra *ServiceAnalyzer* utiliza un *plugin* que invoca a `XMLBeans` con el XML Schema descrito en el paso anterior. A partir del mismo, `XMLBeans` genera y compila las clases e interfaces necesarias. Así, ya podemos tener en memoria los catálogos mientras los creamos mediante instancias de estas clases.

El código generado por este mismo *plugin* se utilizará en el marco de trabajo del grupo UCASE para poder leer mediante Java las salidas de *ServiceAnalyzer*.

4.6.3. Lectura de los ficheros WSDL

Para este cometido los tutores del PFC aconsejaron utilizar `WSDL4J (Web Services Description Language for Java)` [54]. WSDL4J es una herramienta que permite la creación, representación y manipulación de documentos WSDL. No se consideraron más opciones puesto que era ideal para nuestro propósito.

Esta biblioteca ofrece las clases necesarias para poder consultar un documento WSDL. Crear un nuevo documento WSDL o leer uno existente, consiste únicamente en ir utilizando los métodos *accesores* y *mutadores*, respectivamente, de las clases que representan los elementos de un WSDL.

En 4.15 podemos ver un ejemplo de código Java que emplea `WSDL4J` para buscar un puerto dado su nombre en un documento ya creado.

Listado 4.14: Ejemplo de uso de WSDL4J

```
1 String wsdlPath = ”example.wsdl”,
```

```
2         portName = ”portExample”;
3
4
5   WSDLReader wsdlReader = WSDLFactory.newInstance().newWSDLReader();
6   Definition definition = wsdlReader.readWSDL(null, wsdlPath);
7
8   // Toma todo los servicios definidos en <<Definition>>
9   // y todos aquellos de las definiciones importadas
10  Map services = definition.getAllServices();
11
12  Iterator serviceIterator = services.values().iterator();
13
14  boolean found = false;
15
16  // Recorre los servicios
17  while (serviceIterator.hasNext() && !found) {
18          Service service = (Service) serviceIterator.next();
19
20          // Determina si el servicio actual contiene el puerto especificado
21          Port port = service.getPort(portName);
22          if (port != null) {
23                  found = true;
24          }
25  }
```

`WSDL4J` ha servido en el presente PFC para leer los ficheros WSDL de entrada, simplificar el análisis de los mismos y mantener una representación en memoria de sus contenidos durante esta fase de análisis. Con una única lectura del fichero, `WSDL4J` es capaz de generar la representación Java correspondiente al contenido del documento proporcionado y de todos aquellos importados por el mismo.

4.6.4. Lectura y análisis de los XML Schema

La lectura y análisis del sistema de tipos definido por todos los ficheros WSDL y XML Schema necesarios para la generación del catálogo ha sido, sin lugar a dudas, una de las tarea más complicadas (si no la más) de todo el Proyecto, en cuanto a implementación se refiere.

Frente a la facilidad aportada por las bibliotecas explicadas en los apartados anteriores, para el manejo de ficheros XML Schema, el tema se complicaba bastante. Por ejemplo, una persona que no sea experta en WSDL, rápidamente es capaz de hacerse con su lógica usando `WSDL4J`. Si un programador no tiene claro si un servicio tiene un *QName* asociado o un identificador, puede averiguarlo explorando los métodos de la clase `WSDL4J` que representa un servicio WSDL.

Sin embargo, XML Schema es un estándar muy abierto (consecuencia de su diseño "por comité"), con infinidad de posibilidades para las definiciones y declaraciones. Esto provoca que no existan herramientas con la suficiente potencia que se necesitaba en este Proyecto. Por ello, se estudiaron y compararon diversas tecnologías en busca de la que nos ofreciera más facilidades.

La primera opción era utilizar **`WSDL4J`**. La idea era obtener el contenido de la sección «types» de la misma manera que obteníamos los servicios, por ejemplo, al leer los documentos WSDL de entrada, como en 4.15. Una vez hecho esto, había que comprobar qué posibilidades ofrecía `WSDL4J` para la exploración de los tipos definidos.

Listado 4.15: Ejemplo de uso de WSDL4J para obtener un elemento XSD

```
1 WSDLReader wreader = WSDLFactory.newInstance().newWSDLReader();
2             definition = wreader.readWSDL(null, mainWSDLUrl);
3 Types types = definition.getTypes();
4 Element element = types.getDocumentationElement();
```

El problema es que el método *getDocumentationElement* no reconocía el elemento y devolvía `null`.

Investigando por foros, se descubrió que era un problema generalizado, que podía aparecer o no dependiendo del sistema operativo y la versión de la herramienta que se emplearán, lo que no aportaba mucha fiabilidad. Además, aunque leyese bien el tipo o el elemento, `WSDL4J` por sí mismo no ofrecía facilidades para desgranar los tipos al nivel que necesitábamos, sino que a menudo se combinaba con el uso de otras tecnologías como JDOM o Castor. Por todo esto, esta opción quedó totalmente descartada.

La siguiente opción era **XSOM (XML Schema Object Model)** [33]. Se pensó que podría ser adecuada ya que era la biblioteca que utilizaba `BPELUnit` internamente. Además, aparentemente, ofrecía facilidades adicionales para leer un documento XML Schema embebido en la sección «types» de un fichero WSDL (justo nuestro caso). En realidad, resultó que lo hacía a través de XSLT y era el propio programador el encargado de crear las hojas de transformación.

Paralelamente se descubrieron dos nuevas alternativas. La primera de ellas, la biblioteca **`XML Schema Infoset Model`** [10] que emplea IBM dentro de *WebSphere Application Server*. La otra alternativa era reutilizar **`XMLBeans`**. A pesar de que no es el uso común de `XMLBeans`, esta biblioteca posee un conjunto de clases con las que analizar la estructura abstracta de los componentes de un XML Schema, que es el que utiliza internamente para dar soporte a las herramientas que proporciona.

Al comparar estas últimas tres tecnologías, se vio que eran parecidas en cuanto a la potencia, por lo que nos decantamos finalmente por `XMLBeans` por los siguientes motivos:

- `XMLBeans` está mejor documentada que las otras dos alternativas.

- No añadía nuevas dependencias al proyecto, puesto que ya se utilizaba en *ServiceAnalyzer* con otras finalidades.

- La clase SCHEMAINSTACEGENERATOR que emplea internamente la herramienta "xsd2inst" de `XMLBeans` daba una idea de cómo recorrer el sistema de tipos.

4.6.4.1. WSDL2XSDTree

Del estudio de `XSOM` se dedujo que probablemente la mejor manera de leer los XML Schema incrustados en un fichero WSDL era utilizar hojas de transformación. Así que se decidió crear un proyecto aparte con el cometido de, dado un documento WSDL, crear un árbol de esquemas con los tipos y elementos definidos en el mismo. El árbol generado es el que se emplea para buscar y analizar los tipos con `XMLBeans`.

En concreto, la implementación de *WSDL2XSDTree* se ha apoyado en el uso de XSLT.

XSLT (véase [38]) es un lenguaje XML estandarizado por el W3C que define hojas de estilo capaces de transformar una entrada XML a otros formatos. Está diseñado principalmente para transformar un documento XML en otro. Sin embargo, XSLT también permite transformar XML a HTML y a cualquier otro formato basado en texto. El uso de hojas XSLT permite definir las transformaciones de forma declarativa, simplificando mucho la tarea de transformación frente a otros métodos, como el uso del API DOM, o el procesado mediante SAX.

Para llevar a la práctica la transformación es necesario un motor XSLT que aplique una hoja de estilos XSLT a un documento fuente XML para producir el documento transformado. Para *WSDL2XSDTree* se ha utilizado XSLT 2.0 junto con el motor Saxon-B 9.1.1.

El uso de XSLT va ligado al uso de otro estándar del W3C: XPath (XML Path Language) 2.0 [40, 39]. Esta tecnología se usa para construir expresiones que recorran y procesen documentos. Puede usarse integrado como parte de XSLT o de otras tecnologías XML, como XQuery, y también puede usarse de modo independientemente.

Su propósito es permitir identificar fácilmente conjuntos de nodos de un documento XML, siguiendo un modelo de datos similar al del estándar DOM, con ligeras diferencias.

La sintaxis de una expresión XPath es declarativa, usando condiciones que deben cumplir los nodos resultado de dicha expresión. Se pueden filtrar nodos en función de su tipo, nombre (en caso de ser elemento), atributos, descendientes, etc.

4.6.5. Construcción del AST

En §4.5.5.2 ya se han adelantado detalles de la implementación de las clases para la construcción de los nodos AST relacionados con el patrón Visitante, así que únicamente se aclararán un par de puntos más.

La implementación del patrón Visitante que se ha empleado desplaza parte del trabajo (el recorrido de la estructura) al propio visitante, con lo que el aceptante no hace ya nada más que llamar a éste:

Listado 4.16: Implementación del método *aceptar*

```
1 @Override
2 public Object accept(Visitor visitor, Object data) {
3     return visitor.visit(this, data);
4 }
```

Parece evidente pensar que si todos los métodos *aceptar* son iguales debería dejarse la implementación en la clase ASTNODE, y sólo redefinirlo en

las clases en que sea distinto. Sin embargo, comprobamos que en la práctica esto no funcionaba ya que no seleccionaba los nodos concretos. Es decir, siempre se acababa llamando a *visit(ASTNode)*.

En los visitantes se ha incluido la operación *visit(ASTNode)* aunque el método no hace nada. Sencillamente es el caso por omisión y por eso se ha añadido.

4.6.6. Generación de plantillas

Como ya se adelantó, el generador de plantillas implementa la interfaz VISITOR con todos los métodos *visit* asociados a cada uno de los tipos de nodo.

En el listado 4.16 podemos observar que los métodos *visit* reciben además del nodo un objeto *data*. Este parámetro, usual en la implementación del patrón Visitante, se ha utilizado para transmitir hacia abajo en el árbol el nombre de la variable que se está tratando.

Por otro lado, el valor de retorno será la cadena que representa la plantilla generada hasta el nivel del árbol actual.

4.6.7. Generación de declaraciones

El generador de declaraciones también implementa la interfaz *Visitor*. En este caso no se utiliza el parámetro *data* ya que los los tipos se transmiten de abajo hacia arriba, el mismo orden en el que funciona la recursividad usada, por lo que nos basta con devolverlos como valor de retorno.

Existe una única excepción: un nodo elemento pasa su identificador al accept de sus nodos atributos (si los tiene) para construir el nombre de la variable asociada al atributo. El nombre de la variable asociada a un atributo es la concatenación del nombre de elemento al que pertenece el atributo y

el nombre del atributo. De este modo, si hubiese más de un elemento en un mensaje con atributos de igual nombre, se distinguirá sin problemas las declaraciones que corresponden a uno y a otro.

En las declaraciones únicamente aparecen las variables asociadas al elemento raíz y a los distintos atributos, ya que son las que van a dar lugar a diferentes mensajes. El resto de variables, son variables Velocity, necesarias para la construcción de la plantilla (por ejemplo, un contador en un bucle).

Puesto que plantillas y declaraciones se generan de forma separada, la manera elegida para mantener una correspondencia entre los nombres de las variables utilizadas en las plantillas y las declaraciones de las mismas ha sido emplear para ellas una convención a partir de los QName almacenados en los nodos elementos y atributos.

4.6.8. Conversión del catálogo generado al formato XML

Este último paso ha resultado muy sencillo, ya que `XMLBeans` convierte tanto XML a Java como Java a XML, así que basta con utilizar el método *save* que proporciona `XMLBeans` con la instancia SERVICESDOCUMENT creado en memoria e inicializado con los datos del catálogo generado.

4.7. Pruebas y validación

4.7.1. Pruebas en XP

Uno de los pilares de la XP es el proceso de pruebas [4]. XP anima a probar constantemente tanto como sea posible. Esto permite aumentar la calidad de los sistemas reduciendo el número de errores no detectados y disminuyendo el tiempo transcurrido entre la aparición de un error y su detección. También

permite aumentar la seguridad de evitar efectos colaterales no deseados a la hora de realizar modificaciones y refactorizaciones.

XP divide las pruebas del sistema en cuatro grupos:

Pruebas unitarias Son diseñadas por los programadores y su función es la de verificar el código. Comprueban de forma automática la funcionalidad de un conjunto reducido y cohesivo de clases.

Pruebas de aceptación También son conocidas como pruebas funcionales. Están destinadas a evaluar si al final de una iteración se consiguió la funcionalidad requerida, es decir, la descrita en la historia de usuario correspondiente. Son diseñadas por el cliente final, con la ayuda de los miembros especializados en pruebas del equipo de desarrollo, ya que el cliente debe saber lo que espera del programa, pero no tiene por qué entender de diseñar pruebas.

Pruebas de integración En XP, se requiere que todo cambio hecho en una sesión de programación sea integrado inmediatamente en el repositorio central. Por lo tanto, las pruebas de integración se realizan de forma constante. Por supuesto, las pruebas de integración deben ir precedidas por las pruebas unitarias de los módulos modificados.

En este caso son importantes debido a que existe ya otro proyecto en el grupo de investigación que necesita la salida de *ServiceAnalyzer*. Tal y como está configurado el servidor de integración continua actualmente, recompila un proyecto del repositorio, no sólo cuando haya modificaciones (*commits*) en éste, sino también cuando las modificaciones se hacen sobre cualquier dependencia suya. Esto también es útil para cuando las modificaciones se hagan en *WSDL2XSDTree*, ya que *ServiceAnalyzer* podría dejar de funcionar.

Pruebas de implantación Desde el inicio de un proyecto, el sistema debe de implantarse en un entorno similar al real, posiblemente a menor escala, y comprobarse su correcto funcionamiento. A pesar de que es solo una práctica recomendada (que no obligatoria) de XP, ha sido fácil seguirla gracias al sistema de integración continua montado para el grupo de investigación.

4.7.2. Plan de pruebas

4.7.2.1. Alcance

Como ya se ha comentado en repetidas ocasiones, este Proyecto tendrá continuidad dentro del grupo UCASE, por lo que se ha intentado automatizar el mayor número posible de pruebas para facilitar el trabajo en caso de producirse modificaciones o ampliaciones en un futuro.

Sin embargo, al jugar con la traducción de tipos XML Schema y ser éste un estándar tan amplio, es imposible tener casos de prueba para todas las posibles definiciones. Aún así, se ha escogido una muestra bastante representativa y de un tamaño considerable.

No obstante, ciertas validaciones se han realizado manualmente para asegurar la fiabilidad del software sin sobrecargar la construcción del mismo ni comprometer la mantenibilidad de las pruebas por tener casos de prueba demasiado exhaustivos. Sobre todo se han hecho pruebas manuales para comprobar el funcionamiento de la aplicación desde la línea de órdenes.

4.7.2.2. Tiempo y lugar

Las pruebas se han realizado en todo momento durante el desarrollo del PFC, siguiendo el principio de “Flujo” de XP (véase §4.1.3.1). Al final de cada

iteración, se dedicó un tiempo adicional para la realización de pruebas de aceptación manuales.

Conforme se completaba la implementación de una iteración, se ejecutaban las pruebas, que se habían diseñado previamente, siguiendo la técnica TDD (como propone la metodología XP).

A partir de los resultados de estas pruebas se realizaban correcciones y se volvían a ejecutar las pruebas, en el caso de que éstas fallasen o, en caso contrario, se comenzaba la implementación correspondiente a una nueva iteración.

A partir de una cierta iteración en la que ya se generaban plantillas y declaraciones, se implementaron y ejecutaron pruebas para comprobar que la generación se llevaba cabo con éxito con los documentos WSDL de composiciones de servicios de ejemplo con las que experimenta el grupo UCASE. Según el nivel de complejidad de la composición, se incluía en una iteración más temprana o se dejaba para más adelante.

4.7.2.3. Naturaleza de las pruebas

La totalidad de las pruebas usadas son pruebas de caja negra, por ser fáciles de mantener a pesar de cambios en la implementación.

Recordemos que las pruebas de caja negra son aquellas que se centran en lo que se espera de un módulo, es decir, intentan encontrar casos en los que el módulo no se atiene a su especificación. Para ello, se apoyan en la especificación de requisitos del módulo, por lo que también se denominan pruebas funcionales. Conociendo una función específica para la que fue diseñado el producto, se pueden diseñar pruebas que demuestren que cada función está bien resuelta. El probador se limita a suministrarle datos como entrada y estudiar la salida, sin preocuparse de lo que pueda estar haciendo el módulo

por dentro.

Las pruebas de caja negra son útiles en cualquier módulo del sistema pero están especialmente indicadas en aquellos que van a servir de interfaz con el usuario.

El problema con las pruebas de caja negra no suele estar en el número de funciones proporcionadas por el módulo (que siempre es un número muy limitado en diseños razonables), sino en los datos que se le pasan a estas funciones. El conjunto de datos posibles suele ser muy amplio.

A la vista de los requisitos de un módulo, se sigue una técnica algebraica conocida como "clases de equivalencia". Esta técnica trata cada parámetro como un modelo algebraico donde unos datos son equivalentes a otros. Si logramos transformar un rango excesivamente amplio de posibles valores reales en un conjunto reducido de clases de equivalencia, entonces es suficiente probar un caso de cada clase, pues los demás datos de la misma clase son equivalentes. Es importante identificar qué rangos de datos pueden alterar el comportamiento del programa y así definir zonas de trabajo. Es imprescindible pasar pruebas con al menos un dato de cada zona, tanto si el programa debe funcionar como si debe dar un mensaje de error. La experiencia indica, además, que suelen producirse fallos en los bordes de las zonas, por lo que se recomienda probar siempre con datos extremos.

Esta técnica es en la que nos hemos basado a la hora de elaborar las pruebas del sistema.

Los casos de prueba se han definido utilizando el framework `JUnit`, el conjunto de bibliotecas creadas por Erich Gamma y Kent Beck para hacer pruebas unitarias de aplicaciones Java. Cabe destacar el uso de XPath en las pruebas para hacer consultas en las salidas, tanto de *WSDL2XSDTree* como de *ServiceAnalyzer*, que comprueben que los resultados son los esperados.

4.7.3. Diseño de pruebas

Se puede decir, que existen tres elementos fundamentales a probar:

- *WSDL2XSDTree*: ¿transforma correctamente el WSDL de entrada en el correspondiente árbol XSD? ¿Se obtiene un árbol XSD que corresponde a los XML Schema definidos por el documento de entrada sin pérdidas? ¿La salida tiene un formato XML Schema válido? ¿Se trata de un proyecto Java correctamente estructurado y documentado?

- *ServiceAnalyzer*: ¿genera un AST correcto dada una definición XML Schema? ¿Es correcta la traducción de tipos? ¿Es capaz de generar la plantilla a partir del AST? ¿Y las declaraciones de variables? ¿Distingue los estilos RPC (Remote Procedure Call) y *document*? ¿Se genera adecuadamente el catálogo completo de mensajes a partir de la interfaz de los servicios de una composición WS-BPEL? Al igual que en el caso anterior, ¿tiene un nivel mínimo de calidad como proyecto Java?

- La integración entre ambos productos: ¿se realiza con éxito? ¿Se capturan y muestran correctamente los fallos?

En base a esto, cada parte requiere una combinación determinada de varios tipos de pruebas:

WSDL2XSDTree Se van a realizar pruebas automáticas de la generación de los árboles y de si tienen un formato XML Schema válido. Para este último propósito intentaremos compilar el XML Schema resultante. También se van a añadir pruebas particulares de casos especiales a contemplar. La parte de interfaz con el usuario se probará de forma manual.

ServiceAnalyzer Se pueden distinguir cuatro grupos principales de pruebas:

1. La traducción de los elementos/tipos XSD al correspondiente árbol de sintaxis intermedia (AST).

2. La generación de plantillas paramétricas para los mensajes.

3. La generación de las declaraciones de las variables de las plantillas.

4. La generación del catálogo de mensajes a partir de los ficheros WSDL y XSD de composiciones WS-BPEL.

Estos puntos son cruciales para la aplicación, y deben incluir sin falta pruebas automatizadas sobre las propiedades que deben cumplir y el comportamiento que deben seguir.

Con respecto a la interfaz de usuario, las limitaciones de tiempo, combinadas con la dificultad y la escasez de herramientas automatizadas de prueba de las interfaces gráficas obliga a usar pruebas de aceptación manuales por lo pronto. Además, en esta primera versión las formas de interactuar con el usuario están muy restringidas, por lo que su prueba manual es rápida.

Integración La integración es otro aspecto difícil de probar automáticamente, dado que depende en gran medida del entorno empleado por el usuario y de la configuración que haya establecido. En este sentido el uso de un sistema de integración continua ha facilitado las pruebas. De todos modos, la integración se presta bien a pruebas de aceptación manuales sobre una configuración realizada de antemano. Se habrá de probar que la integración reacciona bien a errores de los programas y de su invocación, fallando cuando debe y permitiendo una recuperación rápida.

4.7.4. Especificación de los casos de prueba

4.7.4.1. WSDL2XSDTree

Se ha creado una clase en Java denominada TREETESTBASE que define una serie de funciones necesarias para realizar las pruebas unitarias generales a las que se someterá todas y cada uno de los casos a probar. Estas pruebas son las siguientes:

- `XSDIsGeneratedCorrectly`: Comprueba que WSDLTreeTransformer se ejecuta para la entrada dada, es decir, que el correspondiente árbol XSD se genera sin problemas.

- `XSDTreeCompilesCorrectly`: Comprueba que el fichero XSD raíz se ha generado convenientemente y compila sin errores. Para ello, se emplea el compilador de XMLBeans para documentos XML Schema.

Cada caso de prueba deberá definir una clase Java que extenderá a TREETESTBASE y que contendrá los distintos tests que evaluarán de una manera más especifica la salida generada por el programa. Casa caso de prueba tiene como objetivo comprobar que *WSDL2XSDTree* se comporta como se espera bajo unas condiciones determinadas y es en los test de cada subclase donde se evaluarán estos aspectos.

STANDALONEWSDLTEST

Prueba el caso más simple: un fichero WSDL que contiene el XML Schema incrustado en la sección «types» y que no tiene ningún tipo de «import» (ni WSDL, ni XSD). El fichero a partir del cual genera el árbol se llama “GoogleBridge.wsdl”.

En esta clase incluimos tres pruebas concretas:

- Comprueba que los espacios de nombres de la sección «definitions» del fichero WSDL pasan a formar parte del elemento schema del fichero XSD raíz.
- Comprueba que se han recogido todos los tipos complejos que había en el XML Schema inicial.
- Comprueba que se han recogido todos los elementos globales que había en el XML Schema inicial.

IMPORTOTHERWSDLTEST

Prueba el caso de un fichero WSDL cuya sección «types» está vacía, pero que tiene un «WSDL:import». En este caso el fichero al que hace referencia el «import» tampoco contiene nada en la sección types. El fichero a partir del cual genera el árbol se llama "LoanService.wsdl".

En esta clase se añaden las siguientes pruebas:

- Comprueba que el «WSDL:import» se ha transformado en correctamente en un «XSD:import».
- Comprueba que el fichero al que apunta el nuevo «XSD:import» es otro documento XSD y que además el XML Schema que contiene no posee ninguna definición (está vacío).

IMPORTXSDANDWSDLTEST

Prueba el caso de un fichero WSDL que importa otro WSDL y cuya sección «types» contiene un «XSD:import». Se da el caso de que ambos documentos WSDL importan el mismo fichero XSD. El fichero a partir del cual genera el árbol se llama "shippingPropertiesRedundant.wsdl".

En esta clase se ha incluido una única prueba adicional:

- Comprueba que el «WSDL:import» se ha transformado en correctamente en un «XSD:import».

TRANSITIVERELATIONTEST

Prueba el caso de un fichero WSDL que importa otro WSDL que a su vez importa un fichero XSD que utiliza el primero. Este ejemplo es el mismo que se ha usado en el anterior caso, pero se ha eliminado la redundancia: ahora únicamente uno de los dos documentos WSDL importa el XML Schema. El fichero a partir del cual genera el árbol se llama "shippingProperties.wsdl".

En esta clase no se ha incluido ninguna prueba adicional ya que el fichero WSDL original contiene referencias a tipos del XSD, luego si ha pasado la prueba de compilación, suponemos que todo está correcto.

DIFFTARGETNAMESPACE

Prueba el caso de un fichero WSDL que contiene un XML Schema embebido en la sección «types» con un `targetNamespace` diferente al de la sección «definitions» del WSDL. El fichero a partir del cual genera el árbol se llama "shippingProperties.wsdl".

Para DIFFTARGETNAMESPACE se han realizado las siguientes comprobaciones específicas:

- Comprueba que se ha creado un «import» en el XSD raíz.
- Verifica que se ha generado otro fichero XSD y que el XML Schema incluido en el mismo mantiene su `targetNamespace` original.

Este caso de prueba se diseñó *a posteriori* debido a un fallo en las pruebas de *ServiceAnalyzer* que puso de manifiesto que algo no iba bien en *WSDL2XSDTree*.

Prueba el caso de un WSDL con más de un XML Schema embebido en la sección «types», de entre los que, al menos uno, tiene un `targetNamespace` diferente al del elemento «definitions», por lo que ha de sacarse a un fichero XSD externo al crear el árbol.

SEVERALXMLSCHEMATEST

También se añadió una prueba para el caso de un fichero WSDL que incluye más de un XML Schema embebido en la sección «types», pero todos con idéntico `targetNamespace`, el de el bloque «definitions». En concreto, hay dos XML Schemas, cuyos contenidos deberán concatenarse en el fichero XSD raíz.

Para esta clase se han realizado las siguientes comprobaciones específicas:

- Verifica que se han concatenado los XML Schema en el XSD raíz, comprobando que se ha recogido un componente de cada XML Schema.

DEPSINDIFFERENTFOLDERTEST

Prueba el caso de un fichero WSDL hace referencia importa otros WSDL que están ubicados en diferentes carpetas. El fichero a partir del cual genera el árbol se llama “MetaSearch.wsdl”.

Para DEPSINDIFFERENTFOLDERTEST se han realizado las siguientes comprobaciones específicas:

- Comprueba que se ha creado un «import» en el XSD raíz.

- Verifica que la ruta contenida en el atributo *schemaLocation* del «import» es la esperada.

4.7.4.2. ServiceAnalyzer

EQUALSANDHASHCODETEST

Esta clase implementa las pruebas de los métodos *equals()* y *hashCode()* sobrecargados para los nodos. Por cada tipo nodo probamos que:

- Si dos objetos tienen distinto tipo, *equals()* devuelve falso y *hashCode()* devuelve valores distintos.
- Si dos objetos distintos tienen el mismo contenido, dan el mismo *hashCode()* y *equals()* devuelve verdadero.
- Si dos objetos tienen contenido distinto, ambas funciones coinciden en que son distintos.

UTILSTEST

Aquí enmarcamos un conjunto de pruebas automatizadas que pretenden verificar el correcto funcionamiento de una serie de funciones comunes para el manejo de los elementos de las declaraciones de variables de las plantillas.

De forma resumida, detallamos los aspectos que se prueban:

- Los elementos del atributo `element` se separan por coma (da igual el número de espacios que haya entre medio) y el orden en que se declaran importa.
- Los elementos del atributo `values` se representan igual que los de `element`, pero el orden en que se declaran no importa.
- Si dos objetos «typedef» son distintos pero tienen el mismo contenido (deben tener los mismos atributos y con los mismos valores, a excepción

del valor del atributo `name`), el resultado del método *equals()* devuelve verdadero.

- Si dos objetos «typedef» tienen contenido distinto, *equals()* devuelve falso.

ASTTEST

Comprueba que los árboles de sintaxis intermedia generados son los esperados. Para ello, compara mediante el método *equals()* el árbol generado por el PARSER de *ServiceAnalyzer* y el árbol esperado (inicializado "manualmente"). Todos los casos se prueban para los dos estilos SOAP considerados: *document/literal*, *RPC/literal*.

Batería de pruebas de DECLARATIONSGENBASETEST

Se trata de una superclase que generaliza las pruebas del generador de declaraciones.

- DG_TYPESCORRESPONDENCETEST: Comprueba la equivalencia entre los tipos primitivos y derivados del sistema definido por XML Schema y el sistema de tipos definido para *ServiceAnalyzer*. Hay un caso de prueba por cada tipo.

- DG_SIMPLETYPESTEST: Comprueba que se generan correctamente las declaraciones de los tipos XSD simples definidos por el usuario. Como es imposible probar todas las posibles definiciones, se ha elegido una muestra representativa. Hay casos de pruebas para todas las restricciones, para ejemplos de listas y para comprobar que efectivamente el tipo *union* no está soportado en esta versión.

- DG_COMPLEXTYPESTEST: Se han seleccionado los siguientes casos para probar:

  - Una secuencia con un único elemento simple.
  - Una secuencia con más de un elemento simple.
  - Una secuencia en la que uno de los elementos es otra secuencia.
  - Una secuencia de un elemento simple en la que la secuencia puede aparecer cero o una vez.
  - Una secuencia de un elemento simple que puede aparecer cero o una vez.
  - Una secuencia de un elemento simple en la que la secuencia puede aparecer cero o infinitas veces.
  - Una secuencia de un elemento simple que puede aparecer cero o infinitas veces.
  - Una secuencia con más de un elemento simple en la que la secuencia puede aparecer cero o infinitas veces.
  - Una secuencia con más de un elemento simple en la que cada elemento tiene unas limitaciones de repetición diferentes.
  - No hay diferencia en la declaración de tipos de un atributo y el mismo con valor `default` predefinido.
  - Un atributo con valor `fixed` no debe aparecer en la declaración.
  - En una misma declaración no debe haber dos elementos «typedef» que sólo se distingan en el atributo `name`. Es decir, si dos elementos (una «variable» u otro «typedef») de la declaración utilizan el mismo «typedef» se debe reutilizar el primero definido.

- El contenido mixto y los elementos «choice», «all» y «wildcard» no están soportados en esta versión de *ServiceAnalyzer*.

Batería de pruebas de TEMPLATEGENBASETEST

Se trata de una superclase que generaliza las pruebas del generador de plantillas. Las plantillas, al contener código Velocity, han tenido que ser incluidas en un bloque CDATA. Por tanto, no podemos realizar consultas XPath para analizar el contenido de las plantillas como en el resto de pruebas. Como consecuencia, lo que se hace es recuperar la plantilla y comparar la cadena obtenida con la esperada.

En todos los casos de prueba, además de comprobar que la plantilla es la esperada, se ha comprobado que el código Velocity compila sin errores. Además, en casos particulares se ha realizado una asignación de valores a las variables de las plantillas y se verificado que el resultado es válido contra el XSD del que se había extraído el tipo XML Schema.

- TG_TYPESCORRESPONDENCETEST: Comprueba que las plantillas de los tipos base de XML Schema se generan tal y como se espera. En este caso, no se ha descrito un caso de prueba por cada tipo primitivo y derivado de XML Schema, ya que todos los tipos simples van a tener la misma estructura de plantilla, todos los tipos lista las mismas plantillas, etc.

- TG_SIMPLETYPESTEST: En este caso también se han reducido los casos de prueba con respecto a la generación de declaraciones, siguiendo la técnica de clases de equivalencias ya comentada. Basta con probar un caso de tipo primitivo con restricciones, un tipo simple lista y un tipo simple lista con restricciones.

- TG_COMPLEXTYPESTEST: Se han verificado los siguientes aspectos:

- Una secuencia de un único elemento utiliza la variable asociada al elemento raíz.
- Una secuencia de más de un elemento utiliza la variable asociada al elemento raíz, accediendo al valor del elemento n con el método *get* y $n - 1$ como parámetro.
- Si existe una secuencia dentro de otra, se llama a *get* sucesivamente para acceder a los valores de las variables.
- Tanto los elementos opcionales, como los que se pueden repetir n veces utilizan la directiva VTL `foreach`.
- La combinación de los dos casos anteriores se resuelve correctamente.
- No hay diferencia en la declaración de tipos de un atributo y el mismo con valor *default* predefinido.
- Un atributo con valor `fixed` no debe aparecer en la declaración.
- En una misma declaración no debe haber dos elementos «typedef» que sólo se distingan en el atributo `name`. Es decir, si dos elementos (una variable u otro typedef) de la declaración utilizan el mismo «typedef» se debe reutilizar el primero definido.
- El contenido mixto y los elementos «choice», «all» y «wildcard» no están soportados en esta versión de *ServiceAnalyzer*.

Batería de pruebas CompositionsBaseTest

Esta batería de pruebas era de vital importancia dentro del marco de trabajo del grupo de investigación UCASE. Esto se debe a que el objetivo de este módulo es comprobar que el comportamiento de *ServiceAnalyzer* es el esperado con las composiciones WS-BPEL con las que trabaja el grupo.

WS-BPEL es un lenguaje reciente, por tanto, una de las limitaciones al trabajar con este lenguaje es que existen pocas composiciones de gran tamaño públicamente disponibles.

La mayoría de los ejemplos que encontramos en artículos y en tesis son ejemplos clásicos, como el del préstamos bancario que se incluye en el estándar WS-BPEL de la OASIS [44].

Otros ejemplos disponibles en libros, por ejemplo, son muy sencillos, únicamente pretenden explicar al lector las funcionalidades de este lenguaje a grandes rasgos. Sin embargo, ninguno suele profundizar en detalles específicos de este lenguaje y, menos aún, ejemplos que cumplan una gran parte de las "restricciones estáticas" impuestas por el estándar.

La generación del catálogo de mensajes se ha probado con ejemplos de composiciones de la forja del departamento [12] mediante las siguientes clases de prueba:

- SHIPPINGSERVICEASYNCHRONOUSTEST: Este ejemplo de proceso asíncrono procede de la documentación del estándar y es relativamente simple. Implementa un servicio de envío de paquetes en el que existen dos modos de envío: un único envío con el total de los artículos solicitados o diferentes envíos parciales (a medida que los artículos vayan llegando) hasta que todos los artículos solicitados inicialmente hayan sido enviados.

  Ésta es una de las composiciones modificadas. Se partió de la enviada por el Grupo de Investigación en Ingeniería del Software de la Universidad de Oviedo, con el que se colaboró. La versión recibida no estaba en WS-BPEL 2.0, por lo que la primera modificación fue la actualización de la versión. Por otro lado, la composición no servía para el propósito del grupo ya que no se podía ejecutar con los casos de prueba puesto que

se trataba de un proceso abstracto. Para transformar la composición en ejecutable se llevaron a cabo los siguientes cambios:

- Modificación del espacio de nombres del proceso con: xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
- Introducción de un nuevo *partner*: *Intermediary*. El intermediario envía mensajes al servicio indicándole cuántos artículos estás listos para ser enviados.
- Sustitución de las asignaciones opacas por un «receive» con el mensaje del intermediario.

Además se corrigieron algunos fallos en las variables.

- AUCTIONTEST: La composición *Auction* también procede del estándar WS-BPEL 2.0 y se trata de un proceso asíncrono. Consideramos una subasta de una casa. El proceso recoge la información del comprador y del vendedor de una subasta concreta, reporta los resultados de la subasta a un servicio de registro de subasta y, entonces, envía los resultados del registro al vendedor y al comprador.

  La generación del catálogo de esta composición espera una excepción, ya que no cumple las especificaciones del WS-I 1.1 Basic Profile.

- LOANAPPROVALDOCTEST: La composición *Loan Approval* representa un servicio de préstamo bancario. Los clientes envían peticiones de préstamo junto con su información personal y la cantidad requerida. El servicio de préstamo ejecuta un proceso simple que indicará con un mensaje si se concede o no el préstamo solicitado.

- LOANAPPROVALRPCTEST: El ejemplo *Loan Approval RPC* es una modificación de *Loan Approval* con llamadas a procedimientos remoto. Esta

composición se ha modificado ligeramente para la prueba ya que no respetaba las especificaciones del WS-I 1.1 Basic Profile por un único mensaje, así que se procedió a su corrección.

- LOANAPPROVALEXTENDEDTEST: La composición utilizada como entrada para esta prueba es el resultado del PFC de uno de los alumnos que ha colaborado recientemente con el grupo UCASE. Parte del ejemplo del *Loan Approval* para obtener una composición con una mayor complejidad con el objetivo de poder probar el máximo número de operadores de mutación (de entre los que considera la herramienta `GAmera` hasta el momento) con una única composición de ejemplo.

  Se han descartado la mayor parte de los documento WSDL de esta composición para la prueba de *ServiceAnalyzer*, ya que no respetaban las restricciones del WS-I Basic Profile 1.1.

- METASEARCHTEST: La composición *Meta Search* procede de `BPELUnit` [41]. Es un servicio de un motor de búsquedas por Internet. El cliente solicita búsquedas y el servicio devuelve al cliente los resultados encontrados.

- MARKETPLACETEST: Este ejemplo está basado en el de la Web de ActiveVOS [2]. Es un proceso asíncrono que trabaja con dos peticiones: la del comprador y la del vendedor. Su funcionamiento básico consiste en enviar un mensaje de éxito si el precio solicitado por el vendedor es menor o igual a la cantidad ofrecida por el comprador. Ha sido mejorado por el grupo de investigación para que despliegue y se ejecute en ActiveBPEL, y con un BPTS más completo que incluye comunicaciones asíncronas (*receiveSendAsync* y *sendReceiveAsync*).

- SUMSQUARESTEST: Los ficheros WSDL de entrada de esta prueba recrean un servicio que básicamente realiza un sumatorio de variables al

cuadrado, $\sum_i^n i^2$, que utiliza «forEach» paralelo y variables compartidas.

- TACSERVICETEST: El ejemplo *Tac Service* es una imitación de la orden `tac` de UNIX, que invierte las líneas que se le envían.

- TRAVELRESERVATIONSERVICETEST: Parte del ejemplo extraído de la Web de `NetBeans` [9]. Es un ejemplo definido con enlaces tanto síncronos como asíncronos. Este servicio permite a un cliente realizar la reserva de un viaje: el cliente podrá reservar un billete de avión, alquilar un vehículo y reservar un hotel. Esta composición, aunque está incluida en el conjunto de composiciones probadas, no podrá generarse su catálogo de mensajes hasta la versión *ServiceAnalyzer* 1.1 debido a que utiliza el elemento XML Schema «choice» en los tipos de algunos de los mensajes, cuya traducción se ha aplazado por razones de tiempo.

- ASTROBOOKSTORETEST: ASTRO es una tienda virtual de compra-venta de libros, estilo Amazon. Al igual que con la composición *Loan Approval Extended*, se han descartado aquellos ficheros que no respetaban el estándar para poder contar al menos con el catálogo de mensajes de los que sí.

El resto de composiciones del repositorio de ejemplos con el que trabaja el grupo UCASE no se han probado por ser composiciones derivadas de las anteriores, sin que ello implique un cambio significativo en los WSDL ni los XML Schema, sino, en general, en detalles de WS-BPEL que no afectan a la generación del catálogo.

A este conjunto de pruebas se han añadido dos más, que no son composiciones, sino simples ficheros WSDL diseñados con el objetivo de probar algunos detalles de los dos estilos SOAP, *document* y RPC, que no se han podido valorar en el resto de pruebas:

- SAMPLEDOCCOMPOSITIONTEST: Además de probar que la generación del catálogo se produce sin incidencias, comprueba que:

  - Se tratan de forma adecuada los mensajes vacíos. Este caso está contemplado por el estándar y por tanto, podría darse.
  - En todos los casos, el envoltorio de un mensaje de una operación *document* es el *QName* del elemento XML Schema indicado en el atributo *element* del único elemento «part».

- SAMPLERPCCOMPOSITIONTEST: Realiza las siguientes comprobaciones adicionales:

  - Se tratan de forma adecuada los mensajes vacíos.
  - El envoltorio de los elementos «part» de un mensaje de respuesta de una operación RPC es el nombre de la operación seguido de "Response".
  - El envoltorio de los elementos «part» de un mensaje de solicitud de una operación RPC es el nombre de la operación.
  - La URI del *QName* del envoltorio RPC de toda la operación es la que viene en el atributo `namespace` del elemento «body» que viene dentro del elemento «input» u «output» de la operación. En el caso de que este atributo no esté relleno entonces se toma la URI del *QName* del servicio (el `targetNamespace` del WSDL que contiene al servicio).
  - Los elementos que envuelven a cada parte en el estilo RPC están bajo el espacio de nombres con la URI vacía y el nombre del envoltorio es el valor del atributo `name` del elemento «part».

4.7.5. Validación

En la fase de validación se ha mejorado la calidad del código.

Una de las herramientas en las que nos hemos apoyado para la mejora es `Sonar`. A través de la interfaz de `Sonar` podemos ver de forma detallada los puntos débiles de nuestro proyecto (véase 4.9) como errores potenciales en el código, escasez de comentarios, clases demasiado complejas, escasez de cobertura de las pruebas unitarias, etc.

Lo primero que llama la atención es la sección de "Violations" que nos indica los errores que tiene nuestro código dividido en niveles de gravedad. Esta es una visión muy útil para asegurar que nuestro código está escrito de acuerdo a las buenas prácticas de Java mejorando así en eficiencia, usabilidad y mantenibilidad, fundamentalmente.

Esta pantalla también da información del resultado de los test y de su cobertura; así como del porcentaje de líneas que son comentarios y de líneas duplicadas en el código. Este último dato nos puede servir para darnos cuenta de las zonas de la aplicación que están repetidas y que convendría refactorizar en una única clase.

Entre las correcciones realizadas a partir de las indicaciones `Sonar` podemos citar las siguientes:

- Corte de relaciones cíclicas: había dos paquetes con dependencias cíclicas, por lo que se movieron las clases "conflictivas" de un paquete al otro.

- Uso de STRINGBUILDER: esta clase se ha empleado en el generador de plantillas para optimizar las concatenaciones. Un String es un objeto inmutable, por lo que con cada concatenación estamos creando un objeto nuevo. Si bien en pocos objetos no es importante, a la hora de trabajar

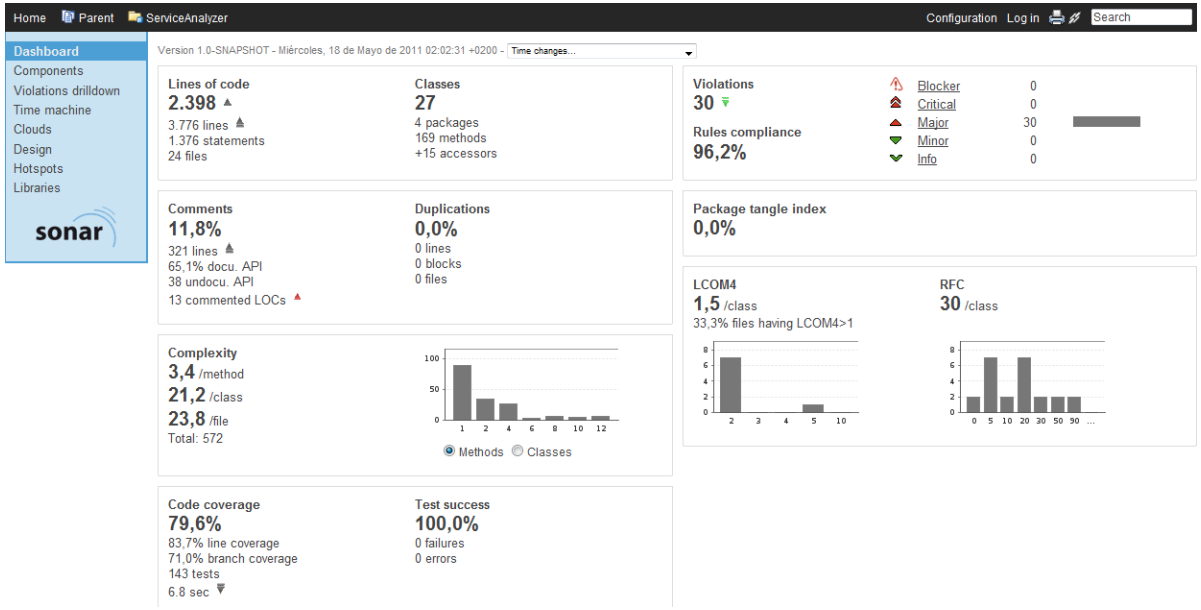

Figura 4.9: Interfaz de Sonar

con muchos String, por ejemplo, cuando la concatenación se produce dentro de un bucle, supondría un despilfarro de memoria. Java provee soporte especial para la concatenación de Strings con la clase STRINGBUILDER. Un objeto STRINGBUILDER es una secuencia de caracteres mutable: su contenido y capacidad puede cambiar en cualquier momento.

Tipo XSD	type	elements	min	max	values	fractionDigits	totalDigits	pattern
string	string							
boolean	string				true,false			
decimal	int							
float	float							
double	float							
duration	duration							
dateTime	dateTime							
time	time							
date	date							
gYearMonth	date							
gYear	date							
gMonthDay	date							
gMonth	date							
gDay	date							
hexBinary	string							[0-9a-fA-F]+
base64Binary	string							
anyURI	string							
QName	string							[a]
NOTATION	string							

Tabla 4.3: Equivalencias entre tipos

[a] ([a-zA-Z_][a-zA-Z_0-9.-]*:)?[a-zA-Z_][a-zA-Z_0-9.-]*

CAPÍTULO 5

Resumen

El “Analizador de Servicios Web basados en WSDL 1.1 para pruebas paramétricas”, como se ha titulado este PFC, ha tenido como resultado un analizador de ficheros WSDL y XML Schema que genera esqueletos de casos de prueba para composiciones de WS (Web Services).

La tarea que realiza el analizador consiste en la identificación de los distintos mensajes que componen el caso de prueba para el programa original, bien como entradas a la composición, bien como respuestas dadas por los distintos servicios invocados.

Como resultado, el analizador producirá un catálogo de plantillas en el lenguaje Apache Velocity. La clave de estas plantillas es que están parametrizadas en base a una serie de variables. Para que la generación de valores válidos para estas variables pueda automatizarse, el catálogo ofrece las declaraciones de tipo de las mismas usando un sistema de tipos simplificado (con respecto a XML Schema) que se ha definido para el caso.

Las plantillas generadas pueden ser usadas como parte de la entrada de `BPELUnit`, el marco de pruebas unitarias de composiciones de Servicios Web

basadas en WS-BPEL 2.0 que está integrado en GAmera.

Este analizador es una pieza fundamental dentro de la extensión que el grupo UCASE quiere hacer a la herramienta GAmera. Se trata de un generador de casos de prueba que complementará la herramienta de generación de mutantes, contribuyendo a mejorar la calidad del conjunto de casos de prueba inicial que se le suministra a dicha herramienta.

Al implementar el analizador como un componente separado, se separa también la generación de casos de prueba de los detalles de WSDL y SOAP. En concreto libera el proceso de generación de casos de prueba de las restricciones que imponen WSDL, SOAP, XML Schema y el WS-I Basic Profile 1.1, que hacen que el proceso sea incómodo y propenso a errores difíciles de depurar.

Aunque la idea original era analizar los servicios de una composición WS-BPEL 2.0 , dado que ServiceAnalyzer parte de ficheros WSDL normales y corrientes, las plantillas Velocity generadas se pueden usar para producir casos de prueba de cualquier Servicio Web.

Un punto fundamental para el éxito de este proyecto ha sido la realización continua de pruebas durante toda su duración. Siguiendo la pautas de XP, se han realizado pruebas de aceptación constantemente, junto con las pruebas automáticas de regresión y de unidad.

La automatización de las pruebas era casi un requisito debido a la continuidad que este Proyecto va a tener dentro del marco de trabajo del grupo UCASE. La principal ventaja de automatizar las pruebas es que aseguran una cierta funcionalidad continuamente sin afectar a la velocidad del desarrollo del proyecto.

Durante el desarrollo del proyecto, se han hecho correcciones en el código a partir de fallos detectados por las pruebas, así como también se ha elaborado

pruebas tras la detección “casual” de un fallo.

CAPÍTULO 6

Conclusiones

6.1. Valoración

La elaboración de este Proyecto ha supuesto un gran aprendizaje y un acercamiento a los proyectos con los que ha de enfrentarse un ingeniero en su vida laboral.

La experiencia de colaborar en un grupo de investigación ha sido muy enriquecedora, no sólo por lo aprendido con el propio Proyecto, sino a través de los seminarios en los que cada uno compartía su trabajo con el resto.

Por lo general, uno de los mayores obstáculos a los que uno se enfrenta en un proyecto de investigación es la escasez de documentación. La bibliografía se reduce a unos pocos artículos y descripciones de estándares duras de leer. En este sentido, el apoyo y los conocimientos aportados por los miembros del grupo han sido de gran ayuda.

La participación en los seminarios ha servido asimismo para mejorar la capacidad para trabajar en equipo y para practicar las intervenciones en público.

Por primera vez he trabajado con un entorno de integración continua, valorando su utilidad y la importancia de tener la última versión de código disponible cuando hay dependencias entre el trabajo de los miembros del equipo. Así he aprendido a utilizar `Maven` para gestionar mis proyectos, a utilizar un sistema de control de versiones como `Subversion`, a valorar la calidad del software en base a las métricas proporcionadas por `Sonar`, etc.

También ha sido la primera vez que utilizaba una metodología ágil en un proyecto de tal envergadura. Aunque algunas de las prácticas recomendadas de la metodología XP no se han podido llevar a rajatabla debido a la compatibilización del PFC con otras actividades y dada la naturaleza de un proyecto de fin de carrera, sí que se ha hecho hincapié en otras, como por ejemplo la programación dirigida a pruebas.

A través del Proyecto en sí, me he iniciado en el ámbito de las arquitecturas SOA y profundizado en la programación de Servicios Web: desde el estudio de composiciones WS-BPEL a la generación de casos de prueba `BPELUnit` (con y sin plantillas Apache Velocity) para dichas composiciones y su ejecución con la herramienta `MuBPEL`, pasando por el estudio de las tecnologías SOAP, WSDL, XML Schema o las especificaciones del WS-I Basic Profile.

En cuanto a Java, aunque ya me había iniciado en el lenguaje, con la implementación del PFC he ganado en soltura y he aprendido a diseñar e implementar pruebas automáticas del código con `JUnit`.

Cabe mencionar también el uso de otros lenguajes y bibliotecas como `XMLBeans`, `WSDL4J`, XSLT, XPath, así como de herramientas tales como `NetBeans` o `XMLEye`.

LaTeX [46, 62, 18] ha sido un elemento muy importante en la elaboración de la documentación del presente Proyecto. Ha sido necesario ampliar los conocimientos que ya se tenían sobre este lenguaje de marcado.

Por último, decir que se ha realizado una importante labor de documentación (*javadocs*, manuales) y que se ha decidido que el proyecto sea Software Libre para que pueda resultar de provecho para la comunidad de desarrolladores.

6.2. Trabajo futuro

Como ya se ha repetido a lo largo de la presente memoria, este Proyecto va a tener continuidad dentro del marco de trabajo del grupo UCASE e incluso abre nuevas líneas de investigación.

Por un lado, se seguirá trabajando en *ServiceAnalyzer* para poder ofrecer la versión 1.1 con las mejoras que se han comentado en §3.5 y que no se han podido introducir, por falta de tiempo, en la versión actual.

Por otro lado, la versión actual podrá ser utilizada para modificar las interfaces de las composiciones WS-BPEL con las que trabaja el grupo. Ya hemos visto que muchas de ellas no respetan las especificaciones del WS-I Basic Profile 1.1 y sería conveniente que lo hicieran. Al ejecutar *ServiceAnalyzer* con los ficheros WSDL correspondientes, el programa mostrará al usuario un mensaje de error describiendo la especificación que no se cumple.

No podemos olvidarnos del origen de este Proyecto: la construcción de un generador de casos de prueba para `GAmera`. *ServiceAnalyzer* es tan sólo una herramienta de la que se servirá este generador en un futuro. Antes de ejecutar el generador de casos de prueba, el sistema analizará el programa original WS-BPEL y la interfaz WSDL correspondiente con *ServiceAnalyzer*, con objeto de analizar los mensajes que componen los casos de prueba.

Ahora queda implementar el generador que se ejecutará a continuación. El objetivo de este nuevo componente es crear nuevos casos de prueba que permitan mejorar la calidad del conjunto de casos de prueba inicial. Estos nuevos

casos de prueba, obtenidos mediante un algoritmo genético, permitirán diferenciar entre los mutantes equivalentes y los mutantes resistentes generados previamente por el generador de mutantes de `GAmera` y clasificados como potencialmente equivalentes.

Hasta el momento en que sea creado el generador de casos de prueba para `GAmera`, sería una buen idea utilizar *ServiceAnalyzer* para ampliar el conjunto de casos de prueba para las composiciones disponibles. Aunque el último paso tenga que hacerse de forma manual, el proceso se agiliza enormemente. De este modo, se podrá avanzar en la investigación de la prueba de mutaciones mientras tanto y además, los alumnos que decidan colaborar en líneas de investigación relacionadas, tendrán una buena batería de ejemplos con los que iniciarse.

CAPÍTULO 7

Manual del usuario

En esta sección se presenta un breve manual de las herramientas de este Proyecto con las instrucciones para la instalación, guías de uso y una breve ayuda para el desarrollador.

7.1. Instalación

ServiceAnalyzer ha sido diseñada con el objetivo de ser incorporada en un futuro al analizador de la herramienta `GAmera`. Sin embargo, en este manual vamos a considerar el caso de que la aplicación se utilice de manera autónoma. Igualmente, a pesar de que *WSDLXSDTree* se utilizará principalmente como dependencia de *ServiceAnalyzer*, aquí describimos cómo instalarla para utilizarla de manera independiente.

7.1.1. Requisitos previos

Se necesita tener instalado un entorno Java compatible con J2SE 5.0 o superior.

Si se utiliza una distribución basada en Debian, se puede probar a instalar el paquete `openjdk-6-jre`. OpenJDK es la iniciativa de Sun, que a fecha de hoy es prácticamente 100 % libre. En Ubuntu 11.04 "Natty Narwhal" se puede instalar una versión de OpenJDK 6.0 para usarla como entorno Java por defecto con estas órdenes:

```
sudo apt-get install openjdk-6-jre

sudo update-alternatives --config java
```

Seguidamente, se escoge la entrada de `openjdk-6-jre` y se pulsa Intro, terminando con este paso. Si se está utilizando openSUSE 10.3, se puede usar sin problemas el entorno J2SE 5.0 de Sun que incluye de fábrica. En caso de que no estuviera instalado por alguna razón, se tendría que instalar los paquetes `java-1_5_0-sun*` a través del gestor de paquetes de YaST.

7.1.2. Instrucciones para la instalación de ServiceAnalyzer

Para instalar *ServiceAnalyzer* es necesario seguir las siguientes instrucciones:

1. Descargar en cualquier directorio la distribución autocontenida. Puede encontrarse en `https://neptuno.uca.es/nexus/content/repositories/releases`

2. Descomprir el tar.gz a cualquiera directorio (`~/bin`, por ejemplo).

   ```
   tar -xzvf service-analyzer-1.0-SNAPSHOT-dist.tar.gz -C
   ~/bin
   ```

3. Crear un enlace simbólico bajo algún directorio del path del usuario al fichero `serviceanalyzer` (que debe tener permiso de ejecución):

```
ln -s ~/bin/service-analyzer-1.0-SNAPSHOT/serviceanalyzer
~/bin/serviceanalyzer
```

7.1.3. Instrucciones para la instalación de WSDL2XSDTree

La instalación de *WSDL2XSDTree* sigue el mismo patrón que la de *ServiceAnalyzer*.

7.2. Uso de la herramienta

7.2.1. Instrucciones de uso para ServiceAnalyzer

Es aconsejable que el usuario tenga algunos conocimientos sobre el intercambio de mensajes en los Servicios Web para que pueda comprender la utilidad de esta herramienta.

Para ejecutar este programa escriba en la terminal el siguiente comando:

```
serviceanalyzer (argumentos)
```

7.2.1.1. Generar el catálogo de mensajes

Consiste en la generación de un documento XML con las plantillas de los mensajes que se intercambian entre la composición y los *mockups*.

La sintaxis es:

```
serviceanalyzer wsdl1...
```

Se debe especificar al menos un fichero WSDL como argumento. Si se especifica más de un fichero WSDL se da por hecho que todos pertenecen a la misma composición, por lo que se genera un único catálogo con todos los mensajes de todas las operaciones de cada servicio descrito en los documentos de entrada.

Para mostrar la ayuda para la generación de catálogos de *ServiceAnalyzer* basta con introducir en la terminal:

```
serviceanalyzer -h
```

ServiceAnalyzer analiza automáticamente los ficheros WSDL importados, por lo que no es necesario suministrarlos como entrada.

Todos los ficheros importados por los documentos de entrada, tanto otros WSDL como los ficheros XSD (si los hay), deben estar en la ruta especificada por los bloques «import» para que la herramienta pueda analizarlos.

7.2.1.2. Analizar una composición WS-BPEL

El lenguaje WS-BPEL permite crear procesos de negocio mediante la composición de Servicios Web preexistentes y ofrecerlos a su vez como WS. WS-BPEL permite especificar la lógica de la composición de los servicios (envío de mensajes, sincronización, iteración, tratamiento de transacciones erróneas, etc.) independientemente de su implementación. El estándar OASIS WS-BPEL 2.0 se ha convertido en la referencia a nivel industrial. Por ello, la importancia económica que están alcanzando las composiciones WS-BPEL obliga a prestar especial atención al caso de que se desee generar el catálogo de mensajes de una composición WS-BPEL.

Para saber qué ficheros se deben proporcionar como entrada a la herramienta, una posibilidad es explorar el fichero BPEL de la composición y examinar aquellos «import» que se corresponden con los ficheros WSDL:

```
<import importType="http://schemas.xmlsoap.org/wsdl/"
 location="..." namespace="..."/>
```

En el atributo `location` de cada «import» tenemos la ruta de cada fichero WSDL utilizado para describir la interfaz de los servicios de la composición, es decir, los que se deben proporcionar como entrada a *ServiceAnalyzer*.

Otra posibilidad es, si se dispone del fichero `BPR`, hacer uso del mismo. `BPR` [1] es la extensión del fichero `JAR` que contiene todos los ficheros necesarios para realizar el despliegue de un proceso de negocio para que pueda ejecutarse en el motor de código abierto ActiveBPEL.

Al descomprimir un fichero `BPR` debe aparecer una carpeta cuyo nombre es WSDL que contiene todos los ficheros que *ServiceAnalyzer* necesita analizar, tanto los WSDL como los XSD. Al ejecutar *ServiceAnalyzer*, únicamente debe indicarse la ruta de los ficheros WSDL que contiene la misma.

Aunque hay instantáneas disponibles del código fuente, éstas son más para los usuarios que los desarrolladores. En caso de querer participar como desarrollador, lo ideal es usar una copia de trabajo del repositorio Subversion. Bastará con instalar el paquete Subversion y seguir algunas instrucciones sencillas.

7.2.2. Instrucciones de uso para WSDL2XSDTree

Es aconsejable que el usuario tenga algunos conocimientos sobre WSDL y XML Schema para que pueda sacar partido de esta herramienta. No obstante, muy posiblemente su uso de manera autónoma irá enfocado a la depuración del proyecto.

Para ejecutar este programa hay que escribir en la terminal el siguiente comando:

[1] Puede generarse con la herramienta `MuBPEL` de `GAmera` a través de la orden:

```
run mifichero.bpts mifichero.bpel
```

```
wsdl2xsdtree (argumentos)
```

Con la siguiente orden se mostrará la ayuda de la herramienta:

```
wsdl2xsdtree -h
```

Para generar el árbol XSD correspondiente al sistema de definiciones y declaraciones XML Schema definidas en un documento WSDL basta con:

```
wsdl2xsdtree wsdl
```

No hay que olvidar que si el fichero WSDL proporcionado como entrada a *WSDL2XSDtree* importa otros ficheros (XSD o WSDL), éstos deben estar en la ubicación indicada por el mismo, aunque no sea necesario especificarlos en la orden.

Si el proceso de creación del árbol finaliza con éxito, en el mismo directorio del fichero de entrada, `nombre_fichero.wsdl`, se encontrará un fichero que es la raíz del árbol de salida, `nombre_fichero.wsdl.xsd`. Analizando el contenido del mismo se podrá comprobar si importa otros ficheros XSD y obtener la ubicación de los mismos.

7.3. Compilación del código fuente

Tras obtener un JDK compatible con J2SE 5.0 o superior, habrá que instalar además la herramienta `Maven` y el entorno de pruebas de unidad `JUnit`, en una de sus versiones 4.X. En Ubuntu 11.04 “Natty Narwhal”, esto se puede hacer mediante:

```
sudo aptitude install openjdk-6-jdk maven junit
```

Ahora se debe crear una copia de trabajo local de la última revisión de la rama principal de desarrollo del repositorio de Redmine:

- **ServiceAnalyzer**

```
svn checkout
https://neptuno.uca.es/svn/sources-fm/service-analyzer
service-analyzer
```

- **WSDL2XSDTree**

```
svn checkout
https://neptuno.uca.es/svn/sources-fm/wsdl2xsdtree
wsdl2xsdtree
```

Ya es posible explorar el proyecto. El código fuente se encuentra en la carpeta `src`. La estructura es idéntica en ambos proyectos y es la siguiente:

`main/java`	Directorio principal que contiene el código Java de la aplicación.
`test/java`	Contiene el código Java de las pruebas.
`main/resources`	Contiene los recursos utilizados por el código principal.
`test/resources`	Alberga los recursos utilizados para las pruebas.
`main/assembly`	Aquí se encuentran los descriptores de distribuciones.

A la hora de desarrollar, será de utilidad aprovechar los objetivos predefinidos de `Maven`. A continuación se exponen los principales:

clean	Elimina los directorios de despliegue del proyecto.
compile	Compila el código fuente del proyecto.

deploy Despliega el paquete resultante en un repositorio central para ser compartido.

install Instala el paquete en el repositorio local para ser usado por otros proyectos locales.

package Crea un paquete con el proyecto a partir de las clases compiladas y recursos.

site Genera un sitio con la documentación del proyecto.

test Ejecuta los test de la aplicación.

Para llevar a cabo alguno de ellos basta con situarse en el directorio del proyecto y ejecutar desde la terminal:

```
mvn objetivo
```

Además, el desarrollo puede hacerse mucho más cómodamente si se emplean los plugins de integración de `Maven` para `Eclipse` o `Netbeans`. Estos plugins permiten usar `Maven` desde una interfaz más amigable, evitando así la línea de órdenes y contando con las funcionalidades de refactorización y notificación de errores y avisos de compilación en directo que ofrecen ambos IDE. Las características añadidas a estos entornos son:

- Construir proyectos `Maven` desde el IDE.
- Gestión de dependencias basadas en la sincronización con el `pom.xml` asociado al proyecto.
- Resolución de dependencias `Maven` en el espacio de trabajo sin necesidad de instalar en repositorios `Maven` locales.

- Descarga automática de las dependencias requeridas desde los repositorios `Maven` remotos.
- Asistentes para crear nuevos proyectos `Maven`, editar los ficheros `pom.xml`, así como para permitir soporte `Maven` en proyectos ya existentes.
- Búsqueda rápida de dependencias en los repositorios remotos de `Maven`.
- Avisos en el editor Java para buscar las dependencias o ficheros `JAR` por el nombre de la clase o del paquete.
- Integración con otras herramientas del IDE de desarrollo elegido.

Los plugins para `Eclipse`, m2eclipse y Eclipse IAM, se pueden encontrar en [24, 23], respectivamente.

En el caso de elegir `NetBeans`, desde la versión 6.7 la integración con `Maven` está incorporada en la instalación estándar. En versiones anteriores basta con instalar el plugin desde el menú (Tools/Plugins).

ANEXO A

WSDL 1.1

WSDL (Web Services Description Language) [58, 42] es un lenguaje basado en XML utilizado para describir la interfaz de WS: qué pueden hacer, dónde se encuentran y qué tipo de datos esperan y en qué formato. WSDL informa sobre cómo se puede interactuar con el Servicio Web pero no dice nada acerca de cómo trabaja.

Podríamos decir que es el manual de operación del WS ya que nos indica cuales son las interfaces que provee y los tipos de datos necesarios para la utilización del mismo.

El hecho de tener que describir el Servicio Web sin dar ningún tipo de información referente a su implementación, ayuda a reducir los problemas de compatibilidad entre dos Servicios Web que ofrecen la misma función, pero que utilizan diferentes implementaciones.

Como adelanto, podemos decir que WSDL considera dos niveles de descripción: uno abstracto y uno concreto (véase figura A.1).

En el nivel abstracto, WSDL define un Servicio Web en términos del mensaje que puede enviar y recibir, normalmente utilizando un formato de datos

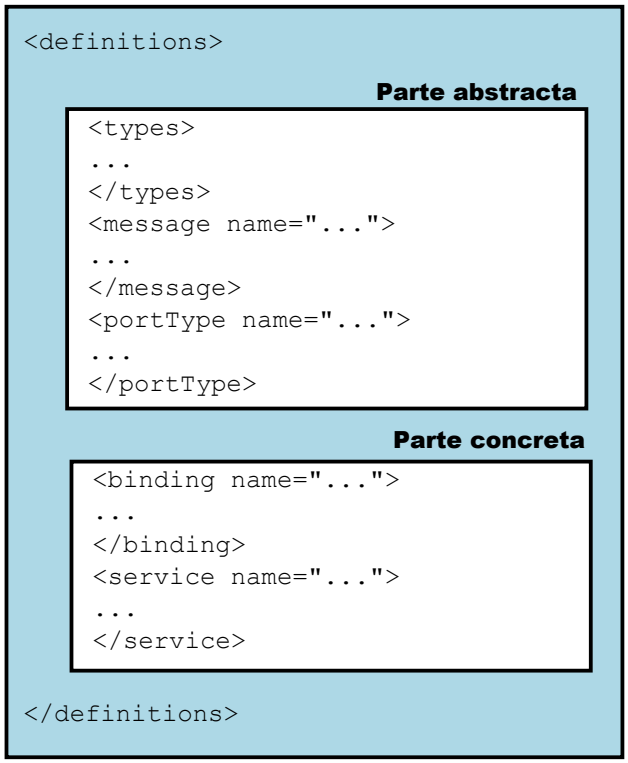

Figura A.1: Estructura de un documento WSDL

basado en XML Schema. Define «operations» que asocian un patrón de intercambio de mensaje (MEP), que puede diferir dependiendo del protocolo de mensaje concreto que se use. También define los «portType» (*interfaces* en WSDL 2.0) que básicamente agrupan las operaciones sin ningún compromiso con un protocolo de transporte.

En el nivel concreto, WSDL define un «binding» que especifica los detalles de transporte y de formato para una o más interfaces. Un «port» (*endpoint* en WSDL 2.0) asocia una dirección de red con un «binding» y, además, una URL (Uniform Resource Locator) a cada servicio. Finalmente, un «service» agrupa los «port» implementando una interfaz común.

A.1. Origen y evolución

La primera versión de WSDL, la 1.0, apareció en septiembre de 2000 y fue desarrollada conjuntamente por IBM, Microsoft y Ariba con el objetivo de describir los Servicios Web para su *SOAP Toolkit*. Se puede decir que WSDL en un lenguaje híbrido ya que se creó combinando los lenguajes propietarios de descripción de servicios de IBM y Microsoft, que son NASSL (Network Accessible Service Specification Language) y SDL (Service Description Language), respectivamente. Del primero, ha heredado la posibilidad para describir llamadas a procedimientos remotos (RPC), usar cualquier tipo de protocolo y cualquier formato de datos. Del segundo ha tomado la posibilidad de usar mensajes.

En marzo de 2001 se publicó WSDL 1.1, que fue en sí una formalización de la versión 1.0, puesto que no introdujo grandes cambios. En esta versión la "D" del acrónimo pasó a ser de "*Definition*" en lugar de significar "*Description*".

No sería hasta junio de 2003 cuando aparecería WSDL 1.2. Según la propia W3C, esta versión, con la que aún trabajan, es mucho más flexible y sencilla para los desarrolladores que las versiones predecesoras. WSDL 1.2 pretende eliminar todas aquellas características que dificultan la interoperabilidad, así como definir un mejor *binding* con HTTP 1.1. Sin embargo, la mayor parte de los servidores SOAP no ofrecieron soporte para WSDL 1.2.

Tras algunos años de cambios, WSDL 1.2 se había convertido en una versión substancialmente diferente a la 1.1, por lo que se renombró a 2.0, convirtiéndose en junio de 2007 en una recomendación de la W3C. Estos cambios eran:

- Añade semántica al lenguaje de descripción
- Elimina los elementos «message», se hace referencia a los tipos directa-

mente desde la definición de «inputs», «outputs» y «faults»

- Los elementos «PortType» pasan a llamarse «Interface»
- Los elementos «Port» pasan a llamarse «Endpoint»

En la figura A.2 podemos observar los cambios estructurales.

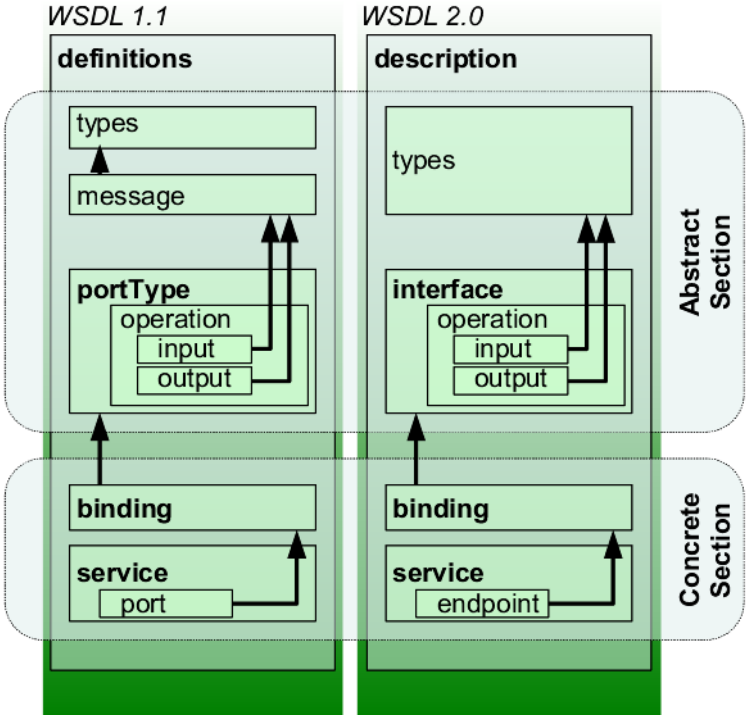

Figura A.2: Comparación entre WSDL 1.1 y WSDL 2.0

A.2. Principios

WSDL se basa en siete principios:

1. **Extensibilidad**. WSDL por sí mismo únicamente permite la descripción del formato de los mensajes, los patrones de interacción y el formato de

conexión entre éstos, y la localización del servicio. No obstante, cuando describimos un servicio, uno puede pensar en diversos atributos que asociar a la descripción del servicio, como por ejemplo, el coste del propio servicio o la calidad de sus acuerdos. WSDL permite la especificación de tales atributos mediante extensiones que serán incluidas en la estructura del núcleo de WSDL.

2. **Soporte para múltiples sistemas de tipos**. Aunque los WS están basados fundamentalmente en XML y XML Schema es el sistema de tipos que predomina en el intercambio de mensajes, WSDL permite el uso de multitud de sistemas de tipos, desde MIME (Multipurpose Internet Mail Extensions) hasta sistemas de tipos dependientes de la plataforma.

3. **Unificación de mensajería y RPC**. Se trata de dos alternativas diferentes al problema de la integración. WSDL fue creado a partir de la combinación de dos lenguajes, uno usa mensajería y otro RPC, por lo que el lenguaje resultante permite ambas alternativas.

4. **Separación de "qué" del "cómo" y "dónde"**. Como podemos observar en la figura A.1, los documentos WSDL tienen dos partes claramente diferenciadas: una parte abstracta con la descripción de lo que hace el servicio en términos generales, y la parte concreta que describe dónde localizar el servicio y cómo llamarlo. Esta separación permite definir un mismo servicio en diferentes localizaciones y usando diferentes métodos de invocación.

5. **Soporte para múltiples protocolos y transportes**. WSDL no es el encargado de dictar los protocolos de comunicación empleados en los mensajes o llamadas enviados a un WS. SOAP es el protocolo con un

uso más generalizado, pero podemos utilizar otro tales como HTTP o SMTP (Simple Mail Transfer Protocol).

6. **Ningún orden**. WSDL no permite la especificación de ningún protocolo de interacción con el servicio. Por ejemplo, impide especificar el orden en el que las operaciones deben ser llamadas. El orden es parte de la semántica (siguiente principio).

7. **Nada de semántica**. WSDL no describe ningún tipo de semántica (como podría ser el significado de una operación, por ejemplo). La semántica debe especificarse en documentos aparte en lenguaje natural o con un lenguaje específico como OWL-S (Ontology Web Language for Services) [1].

Como podemos ver, WSDL permite extensibilidad por todos lados. Esto provoca que existan infinidad de posibles descripciones de servicios, lo que dificulta la interoperabilidad.

A.3. Estructura general de WSDL 1.1

Como ya se ha mencionado, WSDL define los dos niveles de un Servicio Web: un nivel abstracto y un nivel concreto.

Como ya se ha visto en la figura A.1 (página 190), cada nivel es una parte diferenciada en el documento WSDL y ambas se enmarcan dentro del bloque global «definitions». Este bloque puede contener una serie de atributos, entre ellos el «targetNamespace». Puede ser visto como el identificador del WS y, aunque es opcional, se necesita para poder utilizarlo correctamente.

[1] Lenguaje de marcado para anotar semánticamente el contenido de Servicios Web. En concreto, pretende describir qué ofrece el servicio, cómo funciona y cómo se interactúa con él. Se basa en la definición de varias ontologías escritas en OWL que permiten la descripción de servicios web semánticos en diferentes niveles de abstracción.

La parte abstracta está formada por tres secciones: el elemento «types», un número arbitrario de elementos «message» y un número, también arbitrario, de elementos «portType».

Types Contiene las definiciones de los tipos de datos, normalmente utilizando un sistema de tipo XSD, aunque ya hemos comentado que podría emplearse cualquier otro, incluso con un formato diferente al de XML. Los tipos definidos aquí serán referenciados en los elementos «message» para definir el intercambio de información.

Listado A.1: Gramática de «types» utilizando XML Schema

```
1 <definitions .... >
2     <types>
3         <xsd:schema .... />*
4     </types>
5 </definitions>
```

Hay que decir que el contenido de esta sección, en la mayoría de los casos, no es más que un formato intermedio. El formato final de los datos se determinará en la implementación concreta del Servicio Web. Por ejemplo, si un WS se implementa en Java, el sistema de tipos de Java sería el usado.

Message Define el formato de los mensajes que se intercambian entre el Servicio Web y el cliente del mismo. Dicho de otro modo, es el paquete de información que se a envía un WS o que es enviado por éste al cliente o a otro WS. Cada message tiene un nombre único para referenciarlo y está compuesto de una o más partes lógicas, que podrían compararse a los parámetros de la llamada a una función en programación. Cada parte puede referirse a algún elemento de la sección «types» o ser un tipo primitivo del sistema de tipos usado.

Listado A.2: Gramática de «message»

```
1 <definitions .... >
2     <message name="nmtoken"> *
3         <part name="nmtoken" element="qname"? type="qname"?/> *
4     </message>
5 </definitions>
```

PortType Puede definirse como un contenedor de una o más operaciones abstractas que, básicamente, define las operaciones permitidas y la signatura de cada una de ellas. Podría compararse con una biblioteca de funciones o una interfaz.

Operation Una operación normalmente consiste, como en cada lenguaje de programación, en un parámetro o parámetros de entrada, un valor de salida y un elemento de fallo opcional.

Input Define la entrada de la operación, que es un mensaje específico.

Output Define el valor de salida de la operación. El valor es también expresado como un mensaje.

Fault Puede considerarse como un mensaje de excepción que se devuelve desde el Servicio Web al que ha llamado, en caso de que se capture un error. El error está dentro de un mensaje.

Listado A.3: Gramática de «portType»

```
1 <wsdl:definitions .... >
2     <wsdl:portType name="nmtoken">
3         <wsdl:operation name="nmtoken" .... /> *
4     </wsdl:portType>
5 </wsdl:definitions>
```

Una operación WSDL 1.1 soporta cuatro tipos de patrones de intercambio de mensajes:

Sentido único (*one-way*) El *endpoint*, por ejemplo, el Servicio Web, recibe un mensaje y no devuelve nada.

Listado A.4: Gramática de una operación de tipo *one-way*

```
1 <wsdl:definitions .... >
2     <wsdl:portType .... > *
3         <wsdl:operation name="nmtoken" parameterOrder="nmtokens"
              >
4             <wsdl:input name="nmtoken"? message="qname"/>
5         </wsdl:operation>
6     </wsdl:portType >
7 </wsdl:definitions>
```

Petición-respuesta (*request-response*) El «endpoint» recibe un mensaje y devuelve un mensaje correlacionado al que ha llamado.

Listado A.5: Gramática de una operación de tipo *request-response*

```
1 <wsdl:definitions .... >
2     <wsdl:portType .... > *
3         <wsdl:operation name="nmtoken" parameterOrder="nmtokens"
              >
4             <wsdl:input name="nmtoken"? message="qname"/>
5             <wsdl:output name="nmtoken"? message="qname"/>
6             <wsdl:fault name="nmtoken" message="qname"/>*
7         </wsdl:operation>
8     </wsdl:portType >
9 </wsdl:definitions>
```

Solicitud-respuesta (*solicit-response*) El *endpoint* envía un mensaje y recibe una respuesta correlacionada.

Listado A.6: Gramática de una operación de tipo *solicit-response*

```
1 <wsdl:definitions .... >
2     <wsdl:portType .... > *
3         <wsdl:operation name="nmtoken" parameterOrder="nmtokens"
              >
4             <wsdl:output name="nmtoken"? message="qname"/>
5             <wsdl:input name="nmtoken"? message="qname"/>
6             <wsdl:fault name="nmtoken" message="qname"/>*
7         </wsdl:operation>
8     </wsdl:portType >
9 </wsdl:definitions>
```

Notificación (*notification*) El *endpoint* envía un mensaje y no espera respuesta alguna.

Listado A.7: Gramática de una operación de tipo *notification*

```
1 <wsdl:definitions .... >
2     <wsdl:portType .... > *
3         <wsdl:operation name="nmtoken">
4             <wsdl:output name="nmtoken"? message="qname"/>
5         </wsdl:operation>
6     </wsdl:portType >
7 </wsdl:definitions>
```

Hasta aquí la parte del "qué". Con esta información, el cliente potencial del servicio ya sabe cuáles son las operaciones disponibles, qué mensajes y con qué partes debe invocar cada una de esas operaciones y, si es el caso, qué mensajes y con qué partes ha de esperar como respuesta.

No obstante, la descripción es abstracta aún en el sentido de que no se sabe nada acerca de la localización del servicio ni de la manera de conectarse a él, es decir el "dónde" y el "cómo". Todo esta información pertenece a la

parte concreta, la cual puede contener un número arbitrario de elementos «binding» y «service».

Binding Asocia a un grupo de operaciones («portType») una especificación de la codificación de mensajes y el protocolo de transporte (atributo `transport`) a utilizar. El protocolo de comunicación más utilizado es HTTP y el de representación de mensajes entre las partes es SOAP.

En el caso de utilizar SOAP, el modo en que los datos son incluidos en el cuerpo es determinado por el valor de dos parámetros, `style` y `use`, que se corresponden con el estilo y la codificación de SOAP, respectivamente.

Se consideran dos tipos de estilo SOAP para las operaciones:

- Document: los mensajes conllevan toda la información necesaria por lo que son directamente el *body* de SOAP. Todas las partes del mensaje se insertan directamente en el envoltorio SOAP. En sí, las partes del mensaje y sus nombres no juegan ningún papel especial. Normalmente, el tipo de las partes se especifica a través del atributo *element*.
- RPC: los mensajes que siguen este estilo se caracterizan por incluir los parámetros de la llamada al procedimiento remoto. Cada atributo «part» del WSDL de la operación indica un parámetro o un valor de respuesta que se encuentra dentro del cuerpo del mensaje de SOAP. El envoltorio SOAP representa la llamada al procedimiento remoto. Por lo general, el tipo de datos de cada una de las partes se especifica mediante el atributo `type`.

Service Un servicio es un contenedor que agrupa los elementos *port*. Los puertos de un mismo servicio guardan las siguientes relaciones:

- Ningún puerto se comunica con otro (por ejemplo, la salida de uno no es la entrada de otro).
- Si dos o más puertos comparten también «PortType» pero tienen distinto «binding» significa que tienen el mismo comportamiento y por tanto, el consumidor del WS elegirá uno u otro en función del protocolo, la distancia o el formato de los mensajes.
- Examinando los puertos se pueden determinar los tipos de puertos del servicio. Esto permite al consumidor potencial del servicio decidir si comunicarse o no con el mismo en función de si soporta o no varios tipos de puerto.

Port Especifica la dirección, en concreto la URL, para un «binding». Especifica una dirección para el enlace definiendo un único punto de destino de la comunicación, también llamado *endpoint*.

En la figura A.3 podemos observar las relaciones entre los elementos que acabamos de definir.

A.4. Usos

El W3C ha identificado tres usos potenciales de WSDL:

1. Como lenguaje de descripción de interfaz del servicio (IDL). Los WS exponen un sistema software con el que las aplicaciones cliente pueden interaccionar a través de la red. Pero, para que tal interacción se lleve a cabo de forma satisfactoria, el servicio debe ser descrito y publicado a sus consumidores potenciales. Es decir, es necesario establecer una especie de contrato entre servicio y cliente en el que se describa qué hace el servicio (sus operaciones y el formato de los mensajes), cómo in-

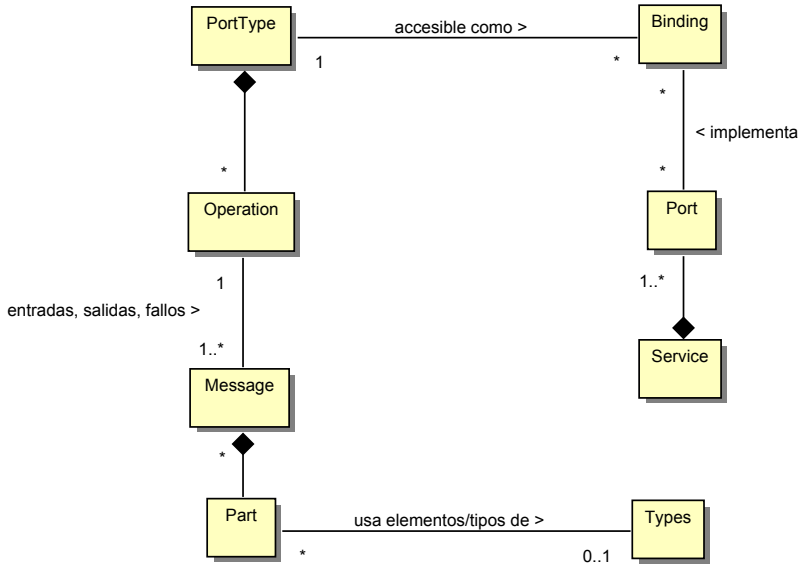

Figura A.3: Relaciones entre los elementos de un documento WSDL

teractuar con él (*bindings* y protocolos) y dónde encontrarlo (URL del *endpoint*).

2. Como entrada para compiladores/generadores de *stubs* y/o *skeleton*. *Stub* es el nombre que recibe el *proxy* en el cliente de un Servicio Web, mientras que se denomina *skeleton* al *proxy* del Servicio Web en lado del servidor.

   Este *proxy* no es más que una clase que se genera, por cada WS, a partir de su descripción WSDL. Contiene la misma interfaz que éste (igual número de métodos y la misma signatura de los mismos). Esta clase se encarga de construir las llamadas SOAP al WS y de recibir las que envía el servicio.

   Las bases de datos comerciales han comenzado a ofrecer funcionalidad para generar automáticamente las descripciones WSDL de procedimientos almacenados. Por otro lado, los servidores de aplicaciones dan faci-

lidades para generar *stubs* de documentos WSDL y para generar WSDL de clases (clases Java, por ejemplo).

3. Existe un tercer uso no definido claramente por el W3C en referencia a la semántica. Sugiere utilizar WSDL para capturar la información que permitirá a los diseñadores razonar sobre la semántica de los servicios. Pero la semántica está fuera en estos momentos de la especificación WSDL, como ya se ha visto, por lo que no tiene sentido.

ANEXO B

Plantillas Velocity

B.1. Introducción

A partir de la versión 1.5 de `BPELUnit` [41], los ficheros BPTS incorporan el lenguaje de plantillas Apache Velocity [21]. Las plantillas permiten generar los mensajes a partir de una fuente de datos y una serie de variables predefinidas. Esto permite que el usuario pueda definir fácilmente diversos casos de prueba que tengan las mismas actividades, pero distinto contenido en los mensajes. Con ello, se obtienen, principalmente, tres ventajas:

- Facilita la generación de los casos de prueba y se hace más sencilla su automatización.
- Se ha separado la generación de casos de prueba de los detalles de WSDL y SOAP.
- Permite la creación de *mockups* más inteligentes, que consideren los mensajes que reciben (qué hay en la petición).

Para ello, se han introducido una serie de cambios en los ficheros BPTS frente a la estructura presentada en §1.5.3.2 que detallaremos en la siguiente sección.

B.2. Novedades de BPELUnit 1.5

Entre ellos se encuentra la aparición de nuevos elementos:

template Este bloque puede sustituir al bloque «data» e indica que lo que viene dentro es una plantilla Velocity donde se pueden usar las variables de `BPELUnit` y las definidas en los casos de prueba.

El objetivo de estas plantillas es construir los mensajes a partir de los casos de prueba.

Las plantillas son unas independientes de otras, de manera que si se asigna una variable en una no tiene efecto en otra.

En «condition» (las restricciones que deben cumplir los mensajes que se reciben) también se pueden utilizar las variables definidas en los casos de prueba.

Otra novedad es el uso de `assume`. Una actividad o un «partner track» entero puede saltarse (no se procesa) si una determinada expresión XPath se evalúa como falsa, incluyendo esta expresión en el atributo `assume` de la actividad o del «partner track» que corresponda. Este atributo puede utilizarse en los siguientes elementos: «partnerTrack», «sendReceive», «receiveSend», «sendReceiveAsynchronous», «receiveSendAsynchronous», «sendOnly», «receiveOnly», «wait». Sin embargo, no está disponible para los siguientes: «clientTrack» [1], «send», «receive».

[1] No se puede emplear en el elemento en sí, sí en las actividades incluidas en el mismo

setUp Los bloques «setUp» son opcionales y se pueden incluir a nivel de «testCase» y/o a nivel de «testCases». En dichos bloques podemos encontrar a su vez:

script Es un bloque de preparación. Se utiliza para añadir una nueva variable o inicializar/cambiar el valor de una ya existente para que se aplique con dicho valor en todos los casos de prueba. El contenido de este bloque será el código Apache Velocity que lo implemente.

dataSource Para generar más de un caso de prueba a partir de de una única plantilla, necesitamos cargar una fuente de datos. La ya citada fuente de datos contiene una serie de filas. Cada fila asigna a cada variable de la fuente de datos un conjunto de valores. Al combinarse con las plantillas, se obtendrán tantos casos de prueba como filas haya en la fuente de datos.

Las fuentes de datos definidas a nivel de caso de prueba tienen precedencia sobre las definidas a nivel del conjunto de casos de prueba, reemplazándolas para este caso de prueba. Las fuentes de datos no se heredan de un caso de prueba a otro.

Este bloque permite varios formatos. El formato usado se especifica a través del atributo `type`. Por su lado, el atributo `src` proporciona la dirección donde encontrar el fuente (una ruta absoluta, una ruta relativa o una URL). Si este atributo no se especifica, se supone que los casos de prueba están incrustados en el fichero, en un elemento hijo denominado «contents». `BPELUnit` busca en un lado o en otro, es decir, que no se pueden definir los dos atributos a la vez. Para que un tipo de formato pueda ser usado debe estar registrado en el registro extensiones de `BPELUnit`.

A continuación se describen los principales tipos de formato que presenta `BPELUnit`:

- *CSV*: Los ficheros CSV (Comma-Separated Values) están formados por filas en la que cada una representa un caso de prueba.

  La propiedad `headers`, si se especifica, contendrá una lista de nombres de variables separados por comas. Si no se especifica esta propiedad, la primera fila contendría los nombres de las variables en lugar de los valores con los que sustituir las variables en el primer caso de prueba. Por ejemplo, en B.1 si quitásemos la línea número 3 habría que añadir "name,amount" al principio del elemento «contents».

  Los campos de cada caso de prueba se pueden separar por comas, espacios o tabulaciones según se indique en la propiedad *separator*. Por defecto, el separador es la tabulación.

Listado B.1: Ejemplo de *dataSource* con formato tipo CSV

```
1 <dataSource type="csv">
2   <property name="separator">,</property>
3   <property name="headers">name,amount</property>
4   <contents>
5     Fred,100
6     Bob,200
7   </contents>
8 </dataSource>
```

- *velocity*: Los ficheros Velocity están formados por filas, cada fila representa los valores de cada una de las variables para cada caso de prueba. La primera componente corresponderá al primer caso de prueba, la segunda al segundo, etc.

Las variables que se vayan a declarar en la fuente de datos cuando el tipo es Velocity obligatoriamente han de listarse (separadas por espacios) en la propiedad `iteratedVars`. Todas las variables que no se declaren aquí pero sí aparezcan en la fuente de datos, únicamente se copiaran tal cual, pero no se sustituirán en las plantillas.

Todas las variables incluidas en `iteratedVars` deben tener asociadas en la fuente de datos una lista de valores con idéntico número de elementos.

Listado B.2: Ejemplo de *dataSource* con formato tipo Velocity

```
1 <dataSource type="velocity">
2   <property name="iteratedVars">lines</property>
3   <contents>
4     #set($lines = [[], ['A'], ['A','B'], ['A','B','C']])
5   </contents>
6 </dataSource>
```

En el ejemplo B.2 podemos ver que hay 4 filas definidas para la variable *lines*. Esto quiere decir que en alguna actividad del BPTS hay una plantilla que utiliza esta variable y que `BPELUnit` probará el mensaje con la lista vacía en el primer caso de prueba, con la lista formada por 'A' en el segundo y así hasta llegar a la ejecución del cuarto caso de prueba con la cuarta fila definida en el `#set`.

- *excel*: La fuente de datos puede cargarse a partir de ficheros Microsoft Excel (.xls o .xlsx). Por defecto, los datos se empiezan a leer desde la primera hoja del fichero. Si se desea que comience en otra diferente, ha de especificarse mediante la propiedad `sheet`, teniendo en cuenta que el valor "0" se corresponde con

la primera hoja. Los nombres de las variables se extraen de la primera línea de la hoja seleccionada.

Puesto que el formato de los BPTS no permiten incrustar ficheros binarios en el elemento «contents», obligatoriamente debe leerse la fuente de datos del fichero externo que se especifique en `src`.

- *ods*: Es idéntico al caso anterior, pero se utilizan hojas de cálculo OpenDocument (.ods).
- *html*: Este tipo de formato lee las fuentes de datos de documentos HTML, que no tienen por qué ser XHTML (eXtensible Hyper Text Markup Language) válidos. Para obtener los datos, se toman las filas de las tablas de la página HTML.

  En la propiedad `table` podemos indicar de qué tabla tomar la fuente de datos. Por defecto su valor es "1", es decir, se lee de la primera tabla.

  Cada celda («td») de la primera fila («tr») es usada para indicar el nombre de la variable. El resto de filas contienen los valores para estas variables en cada caso de prueba.

  En el caso de no utilizar un fichero externo, el código HTML incrustado en «contents» debe incluirse dentro de un bloque CDATA para que no sea analizado como código XML. Esto es especialmente importante en el caso de que el contenido no sea XHTML válido (como en B.3).

Listado B.3: Ejemplo de *dataSource* con formato tipo HTML

```
1 <dataSource type="html">
2   <contents>
3     <![CDATA[
4       <table>
```

```
5           <th>A
6           <th>B
7           <th>C</th>
8         <tr>
9           <td>1
10          <td>2
11          <td>3
12    ]]>
13  </contents>
14 </dataSource>
```

Además de estos tipos, también es posibles definir uno nuevo y registrarlo.

B.2.1. Variables predefinidas

Como ya se ha mencionado, además de las variables Velocity que el usuario cree para los casos de prueba, en las plantillas se pueden utilizar las variables predefinidas que ofrece `BPELUnit`, que son las que se listan a continuación.

B.2.1.1. A nivel del conjunto de casos de prueba

- $baseURL (String): URL base para las URL simuladas.
- $collections (java.util.Collections): métodos útiles para manejar listas y otras colecciones.
- $putName (String): nombre del proceso que se está probando.
- $testCaseCount (int): número total de casos de prueba del conjunto.
- $testSuiteName (String): nombre del conjunto de casos de prueba.

B.2.1.2. A nivel de caso de prueba

- $testCaseName (String): nombre del caso de prueba actual.

B.2.1.3. A nivel de «partner track»

- $partnerTrackName (String): nombre del *partner track* actual.
- $partnerTrackURL (String): URL del *partner track* actual.
- $request (org.w3c.dom.Element): cuerpo SOAP del último mensaje recibido del *partner track* actual.
- $partnerTrackReceived (java.util.List<org.w3c.dom.Element>): lista con todos los mensajes SOAP recibidos por el *partner track* actual.
- $partnerTrackSent (java.util.List<org.w3c.dom.Element>): lista con todos los mensajes SOAP enviados por el *partner track* actual.

B.2.1.4. A nivel de actividades

- $xpath (org.bpelunit.framework.util.control.XPathTool): objeto utilizado para ejecutar consultas XPath o nodos DOM.

B.3. Un ejemplo

En esta sección vamos a presentar un fichero BPTS que pertenece a la misma composición (préstamo bancario) que el del listado B.4 (página 210), pero que utiliza plantillas Velocity. A continuación comentamos las diferencias con respecto el ejemplo sin plantillas.

Listado B.4: Estructura de un fichero *.bpts*

```
1 <?xml version="1.0" encoding="UTF-8"?>
```

```
<tes:testSuite
    xmlns:esq="http://xml.netbeans.org/schema/Loans"
    xmlns:ap="http://j2ee.netbeans.org/wsdl/ApprovalService"
    xmlns:as="http://j2ee.netbeans.org/wsdl/AssessorService"
    xmlns:sp="http://j2ee.netbeans.org/wsdl/LoanService"
    xmlns:pr="http://enterprise.netbeans.org/bpel/N6_ServicioPrestamo/
        LoanApprovalProcess"
    xmlns:tes="http://www.bpelunit.org/schema/testSuite">

  <tes:name>LoanServiceTest</tes:name>
  <tes:baseURL>http://localhost:7777/ws</tes:baseURL>

  <tes:deployment>
    <tes:put name="LoanApprovalProcess" type="activebpel">
      <tes:wsdl>LoanService.wsdl</tes:wsdl>
      <tes:property name="BPRFile">LoanApprovalDoc.bpr</tes:property>
    </tes:put>
    <tes:partner name="assessor" wsdl="AssessorService.wsdl"/>
    <tes:partner name="approver" wsdl="ApprovalService.wsdl"/>
  </tes:deployment>

  <tes:testCases>
    <tes:testCase name="MainTemplate" basedOn="" abstract="false" vary="false">
      <tes:setUp>
        <tes:script>
          #set( $integer = 0 )
        </tes:script>
        <tes:dataSource type="velocity">
          <tes:property name="iteratedVars">
            req_amount ap_reply ap_limit as_reply as_limit accepted
          </tes:property>
          <tes:contents>
#set($req_amount = [ 150000, 150000, 1500, 1500, 1500, 3000, 6000, 8000, 8000])
#set($ap_reply = [ 'true', 'false','silent','true','false','silent','false','smart','smart'])
```

```
35 #set($ap_limit = [ 0, 0, 0, 0, 0, 0, 0, 12000, 6000])
36 #set($as_reply = ['silent','silent', 'low','high', 'high', 'smart','smart','smart','smart'])
37 #set($as_limit = [ 0, 0, 0, 0, 0, 5000, 5000, 5000, 5000])
38 #set($accepted = [ 'true', 'false', 'true','true','false', 'true','false', 'true','false'])
39             </tes:contents>
40           </tes:dataSource>
41         </tes:setUp>
42
43         <tes:clientTrack>
44           <tes:sendReceive
45               service="sp:LoanServiceService"
46               port="LoanServicePort"
47               operation="grantLoan">
48             <tes:send fault="false">
49               <tes:template>
50                 <esq:ApprovalRequest>
51                   <esq:amount>$req_amount</esq:amount>
52                 </esq:ApprovalRequest>
53               </tes:template>
54             </tes:send>
55             <tes:receive fault="false">
56               <tes:condition>
57                 <tes:expression>esq:ApprovalResponse/esq:accept</tes:expression>
58                 <tes:value>$accepted</tes:value>
59               </tes:condition>
60             </tes:receive>
61           </tes:sendReceive>
62         </tes:clientTrack>
63
64         <tes:partnerTrack name="approver">
65           <tes:receiveSend
66               service="ap:ApprovalServiceService"
67               port="ApprovalServicePort"
68               operation="approveLoan"
```

```
69                assume="$ap_reply != 'silent'">
70              <tes:send fault="false">
71                <tes:template>
72                  <esq:ApprovalResponse>
73  #set( $amount = $integer.parseInt($xpath.evaluateAsString("//esq:amount", $request)) )
74  #if( $ap_reply != 'smart' )
75                      <esq:accept>$ap_reply</esq:accept>
76  #elseif( $ap_limit > $amount )
77                      <esq:accept>true</esq:accept>
78  #else
79                      <esq:accept>false</esq:accept>
80  #end
81                  </esq:ApprovalResponse>
82                </tes:template>
83              </tes:send>
84
85              <tes:receive fault="false">
86                <tes:condition>
87                  <tes:expression>//esq:amount</tes:expression>
88                  <tes:value>$req_amount</tes:value>
89                </tes:condition>
90              </tes:receive>
91            </tes:receiveSend>
92          </tes:partnerTrack>
93
94          <tes:partnerTrack name="assessor" assume="$as_reply != 'silent'">
95            <tes:receiveSend
96                service="as:AssessorServiceService"
97                port="AssessorServicePort"
98                operation="assessLoan">
99
100             <tes:receive fault="false">
101               <tes:condition>
102                 <tes:expression>//esq:amount</tes:expression>
```

```
103                 <tes:value>$req_amount</tes:value>
104               </tes:condition>
105             </tes:receive>
106
107             <tes:send fault="false">
108               <tes:template>
109                 <esq:AssessorResponse>
110 #set( $amount = $integer.parseInt($xpath.evaluateAsString("//esq:amount", $request)) )
111 #if( $as_reply != 'smart' )
112                   <esq:risk>$as_reply</esq:risk>
113 #elseif( $as_limit > $amount )
114                   <esq:risk>low</esq:risk>
115 #else
116                   <esq:risk>high</esq:risk>
117 #end
118                 </esq:AssessorResponse>
119               </tes:template>
120             </tes:send>
121           </tes:receiveSend>
122         </tes:partnerTrack>
123
124       </tes:testCase>
125     </tes:testCases>
126 </tes:testSuite>
```

Como podemos observar, hasta la línea 22 el contenido del fichero es idéntico al del ejemplo sin plantillas. Los cambios comienzan en la línea 24 con la introducción del elemento «setUp» que define la fuente de datos: variables a utilizar (línea 30) y conjunto de valores que tomarán las mismas para cada caso de prueba (líneas 33 a 38). Por ejemplo, la cantidad de dinero solicitada por el cliente se representa con la variable *$req_amount* y en los dos primeros casos de prueba se sustituirá por 150.000. Al ejecutar el BPTS

contra la composición, `BPELUnit` sustituirá las variables en las plantillas por cada columna de valores, generando y ejecutando tantos casos de prueba diferentes como columnas haya. En este caso, la primera columna de valores se corresponde con el caso de prueba descrito en la página 210, líneas 23 a 65.

La siguiente diferencia viene en los «send» donde el el elemento «data» se ha sustituido por la plantilla («template»). Por ejemplo, en las líneas de la 50 a la 52 tenemos la plantilla correspondiente al mensaje de petición del préstamo por parte del cliente. Ahora la cantidad solicitada no se especifica directamente sino que se utiliza la variable *$req_amount*.

En las plantillas usadas en los «send» del aprobador y del asesor la cosa se complica un poco más.

La respuesta del aprobador se representa mediante la variable *$ap_reply*. Para mostrar hasta que punto podemos automatizar la generación de casos de prueba usando plantillas, se ha considerado el valor "smart". Cuando *$ap_reply* tiene este valor, se ha parametriza el valor de la cantidad límite a partir de la cual el aprobador decide rechazar el préstamo a través de la variable *$ap_limit*. Por ello, se ha introducido un condicional en la plantilla Velocity (líneas 74 a 80), que en el caso de decir que se utilice un límite diferente, simule la decisión del aprobador y genere un mensaje u otro.

La directiva `#set` de la línea 73 tan sólo toma el valor correspondiente a la cantidad recibida del cliente y se lo asigna como entero a la variable local *$amount* para poder comparar el valor con el límite.

En las líneas de la 111 a la 117, podemos ver que en el lado del asesor se ha actuado del mismo modo. Se genera el mensaje (que se mandará al cliente) directamente con el valor que toma la variable (*$as_reply*), a menos que ésta tenga valor "smart", en cuyo caso, se envía verdadero o falso en función del valor que tenga la variable *$as_limit*.

Como ya vimos al describir la composición *Loan Approval*, no siempre el aprobador ha de dar una respuesta al cliente y no siempre el asesor es invocado. Para controlar esto de manera automática se ha introducido la variable *$as_reply* y el atributo `assume` (líneas 69 y 94, respectivamente). Cuando la variable toma valor "silent" la actividad («receiveSend») o el *partner* no se incluyen en el caso de prueba en ejecución.

En conclusión, con este código podemos probar de forma automática el caso de prueba del ejemplo sin plantillas y ocho más gracias a la introducción de código Velocity.

ANEXO C

Creative Commons Public License

Creative Commons

Attribution 3.0 Unported

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

C.1. License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE

LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

C.1.1. Definitions

a. "**Adaptation**" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b. "**Collection**" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contri-

butions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

c. "**Distribute**" means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

d. "**Licensor**" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

e. "**Original Author**" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

f. "**Work**" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a

process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. "**You**" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. "**Publicly Perform**" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. "**Reproduce**" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

C.1.2. Fair Dealing Rights

Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

C.1.3. License Grant

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d. to Distribute and Publicly Perform Adaptations.

e. For the avoidance of doubt:

    i. **Non-waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory

or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii. **Waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

iii. **Voluntary License Schemes**. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

C.1.4. Restrictions

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work

that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(b), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(b), as requested.

b. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the

URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4 (b) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

c. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would

be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

C.1.5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

C.1.6. Limitation on Liability

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

C.1.7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

C.1.8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder

of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

C.2. Creative Commons Notice

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of this License.

Creative Commons may be contacted at `http://creativecommons.org/`.

Referencias

[1] *IEEE Standard for Software Configuration Management Plans*. (828), 1998.

[2] ActiveVOS. *ActiveBPEL WS-BPEL and BPEL4WS engine*. `http://sourceforge.net/projects/activebpel502/`, octubre 2009.

[3] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[4] Kent Beck y Cynthia Andres. *Extreme Programming Explained - Embrace Change*. Addison-Wesley Professional, segunda edición, 2004.

[5] Joshua Bloch. *Effective Java: Programming Language Guide*. Prentice Hall, Birmingham, UK, UK, 2008. ISBN 0321356683, 9780321356680.

[6] M. Canfora, G. y Di Penta. *SOA: Testing and Self-Checking*. *International Workshop on Web Services Modeling and Testing*, páginas 3–12, 2006.

[7] Ben Collins-Sussman, Brian W. Fitzpatrick, y C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, primera edición, 2004. Disponible gratuitamente en `http://svnbook.red-bean.com/nightly/en/index.html`.

[8] Tom Copeland. *PMD*, febrero 2009. URL http://pmd.sourceforge.net/.

[9] Oracle Corporation. *NetBeans*, marzo 2011. URL http://www.netbeans.org/.

[10] Shane Curcuru. *Analyzing XML schemas with the Schema Infoset Model*, julio 2002. URL http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/org.eclipse.xsd.doc/references/articles/dwtip1-scpw/index.html.

[11] Laboratorio Nacional de Calidad del Software de INTECO. *Curso de desarrollo ágil*, junio 2009. URL http://emprendecaminos.com/docs/%5BAgil%5D-Curso%20de%20Desarrollo%20%C3%81gil.pdf.

[12] Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Cádiz. *Instancia Redmine en Neptuno*, 2011. URL https://neptuno.uca.es/redmine/.

[13] Mark Doliner. *Cobertura*, marzo 2010. URL http://cobertura.sourceforge.net/.

[14] Juan José Domínguez-Jiménez, Antonia Estero-Botaro, Antonio García-Domínguez, y Inmaculada Medina-Bulo. *GAmera: An Automatic Mutant Generation System for WS-BPEL Compositions*. En *ECOWS 2009: Seventh IEEE European Conference on Web Services*, páginas 97–106. IEEE Computer Society, Eindhoven, The Netherlands, 2009.

[15] Juan José Domínguez-Jiménez, Antonia Estero-Botaro, y Inmaculada Medina-Bulo. *A framework for mutant genetic generation for WS-BPEL*. páginas 229–240, 2009.

[16] J. J. Domínguez Jiménez, A. Estero Botaro, I. Medina Bulo, M. Palomo Duarte, y F. Palomo Lozano. *El reto de los servicios web para el software libre*. *Proceedings of the FLOSS International Conference*, 2007.

[17] Antonia Estero Botaro, Francisco Palomo Lozano, y Inmaculada Medina Bulo. *Mutation Operators for WS-BPEL 2.0*. En *ICSSEA 2008: Proceedings of the 21th International Conference on Software & Systems Engineering and their Applications*. 2008.

[18] Alexander Feder. *BibTeX*, marzo 2011. URL `http://www.bibtex.org/`.

[19] Apache Software Foundation. *Apache License, Version 2.0*, enero 2004. URL `http://www.apache.org/licenses/LICENSE-2.0.html`.

[20] The Apache Software Foundation. *XMLBeans*, diciembre 2009. URL `http://xmlbeans.apache.org/`.

[21] The Apache Software Foundation. *Apache Velocity*, noviembre 2010. URL `http://velocity.apache.org/engine/devel/`.

[22] The Apache Software Foundation. *Apache Maven*, marzo 2011. URL `http://maven.apache.org/`.

[23] The Eclipse Foundation. *Eclipse IAM (Integration for Apache Maven)*, may 2011. URL `http://www.eclipse.org/m2e/`.

[24] The Eclipse Foundation. *Maven Integration (m2e)*, may 2011. URL `http://www.eclipse.org/m2e/`.

[25] Martin Fowler. *Continuous Integration*. mayo 2006.

[26] Erich Gamma. *JUnit*, diciembre 2009. URL `http://junit.sourceforge.net/`.

[27] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Patrones de Diseño*. Addison-Wesley, 2003.

[28] Antonio García Domínguez. *Instancia Jenkins en Neptuno*, mayo 2011. URL `http://jenkins-ci.org/`.

[29] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Yves Lafon, Jean-Jacques Moreau, Anish Karmarkar, y Henrik Frystyk Nielsen. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C recommendation, W3C, abril 2007. URL `http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`.

[30] Andy Hunt y Dave Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Bookshelf, 2003.

[31] IBM. *Common Public License Version 1.0 (CPL)*, mayo 2005. URL `http://www.ibm.com/developerworks/library/os-cpl.html`.

[32] Dos Ideas. *Maven*, abril 2011. URL `http://www.dosideas.com/wiki/Maven`.

[33] Java.net. *XML Schema Object Model (XSOM)*, enero 2011. URL `http://xsom.java.net/`.

[34] Ron Jeffries. *Misconceptions about XP*, enero 2002. URL `http://xprogramming.com/articles/misconceptions/`.

[35] Ron Jeffries. *XPrograming.com, An agile Software Development Resource*, enero 2002. URL `http://xprogramming.com/`.

[36] JUnit.org. *Resources for Test Driven Development*, mayo 2011. URL `http://www.junit.org/`.

[37] Kohsuke Kawaguchi, R.Tyler Croy, y Andrew Bayer. *Jenkins CI, an extensible continuous integration engine*, mayo 2011. URL http://jenkins-ci.org/.

[38] Michael Kay. *XSL Transformations (XSLT) Version 2.0*. W3C recommendation, W3C, enero 2007. URL http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[39] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer)*. Wrox Press Ltd., Birmingham, UK, UK, 2008. ISBN 0470192747, 9780470192740.

[40] Michael Kay, Don Chamberlin, Jonathan Robie, Mary F. Fernández, Jérôme Siméon, Scott Boag, y Anders Berglund. *XML Path Language (XPath) 2.0*. W3C recommendation, W3C, enero 2007. URL http://www.w3.org/TR/2007/REC-xpath20-20070123/.

[41] Daniel Lübke y Antonio García Domínguez. *BPELUnit - The Open Source Unit Testing Framework for BPEL*, noviembre 2010. URL http://www.se.uni-hannover.de/forschung/soa/bpelunit/.

[42] Philip Mayer. *Design and Implementation of a Framework for Testing BPEL Compositions*. Master's Thesis, Leibniz Universität Hannover, Fakultät für Electrotechnik und Informatik, septiembre 2006.

[43] Philip Mayer y Daniel Lübke. *Towards a BPEL unit testing framework*. En *TAV-WEB 2006: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, páginas 33–42. ACM, 2006. ISBN 1-59593-458-8. URL http://doi.acm.org/10.1145/1145718.1145723.

[44] OASIS. *Web Services Business Process Execution Language 2.0*, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[45] OASIS. *Universal Description, Discovery, and Integration*, 2010. URL http://uddi.xml.org/.

[46] Tobias Oetiker, Hubert Partl, Irene Hyna, y Elisabeth Schlegl. *The Not So Short Introduction to LaTeX2ε*, diciembre 2010. URL http://tobi.oetiker.ch/lshort/lshort.pdf.

[47] The University of Maryland. *FindBugs - Find Bugs in Java Programs*, agosto 2009. URL http://findbugs.sourceforge.net/.

[48] Roger S. Pressman. *Ingeniería del Software: Un Enfoque Práctico*. McGraw-Hill, sexta edición, 2005.

[49] R. Rice. *Surviving the top 10 challenges of software test automation. CrossTalk: The Journal of Defense Software Engineering*, páginas 26–29, 2002.

[50] Ruey-Shun Chen Shien-Chiang Yu. *Web Services: XML-based system integrated techniques*. *The Electronic Library*, 21:358–366, 2003.

[51] Ian Sommerville. *Ingeniería del Software*. Addison-Wesley, sexta edición, 2002.

[52] Sonar. *Sonar*, marzo 2011. URL http://www.sonarsource.org/.

[53] Sontype. *Why Nexus*, marzo 2011. URL http://nexus.sonatype.org/why-nexus.html.

[54] SourceForge.net. *The Web Services Description Language for Java Toolkit (WSDL4J)*, noviembre 2010. URL http://sourceforge.net/projects/wsdl4j/.

[55] Eric T. Freeman, Elisabeth Robson, Bert Bates, y Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 2004.

[56] Checkstyle Development Team. *Checkstyle 5.3*, sep 2010. URL http://checkstyle.sourceforge.net/.

[57] ThoughtWorks. *CI Feature Matrix*, marzo 2011. URL http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix.

[58] Asir S. Vedamuthu. *Web Services Description Language (WSDL) Version 2.0 SOAP 1.1 Binding*. W3C note, W3C, junio 2007. URL http://www.w3.org/TR/2007/NOTE-wsdl20-soap11-binding-20070626.

[59] W3C. *XML Schema*, diciembre 2009. URL http://www.w3.org/XML/Schema.

[60] W3C. *Extensible Markup Language (XML)*, abr 2011. URL http://www.w3.org/XML/.

[61] W3C. *HTTP - Hypertext Transfer Protocol*, marzo 2011. URL http://www.w3.org/XML/.

[62] Peter Wilson. *The Memoir Class*, marzo 2011. URL http://ftp.gui.uva.es/sites/ctan.org/macros/latex/contrib/memoir/memman.pdf.

[63] WS-I. *Basic Profile Version 1.1*, 2006. URL http://www.ws-i.org/Profiles/BasicProfile-1.1.html.